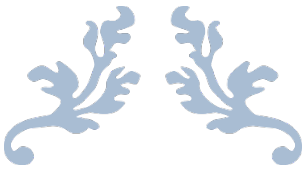


DESIGN REPORT OF FIFA GUI SYSTEM

Shengjie Xia

2023-1-17

LANZHOU UNIVERSITY

Lanzhou, Gansu, China

Abstract

In this project, I created a graphical interface query system for the World Cup tournament. I used *PyQt5* library module for graphical interface design, *socket* library for network connection and data transfer, *sqlite3* library for data storage, *re* library for input format correctness, *os* library for file path reading and *sys* library for process exit. I also use basic common programming methods such as **multi-threading**, **object-oriented programming**, **string manipulation**, and **custom functions**.

I also designed a web crawler file, using the *webdriver* module in the *selenium* library, while using the *pyautogui* library that can control the cursor movement, with the *re* library to get the content of the web page and the time library to simulate the response time, these do assist in crawling.

Since the web crawler requires third-party libraries and corresponding drivers, the environment is more complicated to configure, so I separate the web crawler files from the server-side files, and the crawled data is manually added into the database by me.

Keywords: multi-threading, object-oriented programming, network connection, web crawler, data storage

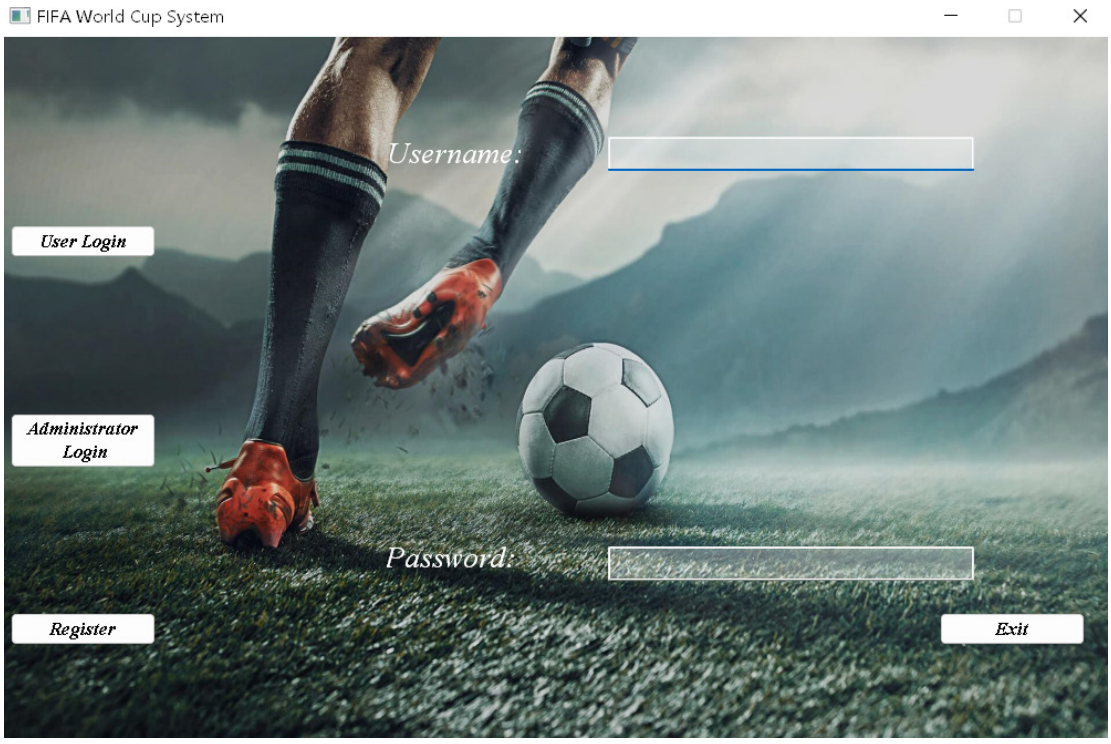


Figure 1: Index of the system

Content

System Design

To build such a system, I created the following files.

administrator_interface.py	2023-01-14 12:53	Python File	16 KB
client.py	2023-01-13 11:21	Python File	2 KB
main_interface.py	2023-01-14 12:50	Python File	10 KB
register_interface.py	2023-01-14 12:55	Python File	10 KB
server.py	2023-01-13 12:17	Python File	28 KB
Source_rc.py	2023-01-15 10:38	Python File	8,730 KB
user_interface.py	2023-01-14 15:01	Python File	24 KB

Figure 2: Documents of the Project

To launch the system, we need click on two files-*client.py* and *server.py*. First let's talk about the design of the interface.

Figure 3: Elements of the Login Interface

As shown in the figure, these are the elements of the login screen. As shown in the figure, these are the elements of the login screen. According to the PyQt5 library itself, we define this window as a class. In PyQt5, the three classes *QMainWindow*, *QWidget* and *QDialog* are used to create windows. Here we choose to inherit the class QMainWindow, mainly for the reason of jumping to the window later. After instantiating this class, it will run as the main window, and only in this way can we completely end the original window after jumping, while if we jump in other ways, the original process will still occupy resources in the background.

So, we first need to do the following.

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import QMainWindow


class UIMainWindow(QMainWindow):

    def __init__(self):
        super().__init__()
        self.setupUi(self)
```

The class method *setupUi()* here is to display the elements we need to layout.

```
    def setupUi(self, FIFASystem):
        FIFASystem.setObjectName("FIFASystem")
        FIFASystem.resize(1007, 634)    # Interface size settings
        FIFASystem.setMaximumSize(1007, 634)
        FIFASystem.setMinimumSize(1007, 634)
        FIFASystem.setStyleSheet("QWidget#FIFASystem{\n"
                                 "        border-image: url(:/GUI Project/FIFA
Background Image.png);\n"
                                 "}")    # Background image settings
```

The second parameter of the function setupUi() is the name of the window object, where all windows I have unified for FIFA System, for each window we have a common operation is to set the size of the window, object name and reference to the style sheet to set the background image.

Next is the setting of each element, first of all, the input box settings.

```
        self.username_line = QtWidgets.QLineEdit(FIFASystem)
        self.username_line.setGeometry(QtCore.QRect(550, 90, 331, 31))
        self.username_line.setStyleSheet("background-color: rgb(255, 255, 255, 0.3);")
        self.username_line.setObjectName("username_line")
```

The settings for the input box are similar, starting with a reference to the *QLineEdit* module, and then setting its position and size. The four parameters passed in the *QRect* are (x-axis coordinate, y-axis coordinate, length, width). Here, from left to right is the positive x-axis direction, and from top to bottom is the positive y-axis direction. When setting the background color in the reference style sheet, the last parameter is transparency (between 0 and 1, 1 means completely transparent, 0 means opaque).

```
        self.password_line.setEchoMode(QtWidgets.QLineEdit.Password)
```

For the password input box, we also need to hide the password.
For the button setting, we need to set some properties of the font.

```
self.register_button = QtWidgets.QPushButton(FIFASystem)
self.register_button.setGeometry(QtCore.QRect(10, 520, 131, 29))
font = QtGui.QFont()
font.setFamily("Times New Roman")
font.setPointSize(10)
font.setBold(True)
font.setItalic(True)
font.setWeight(75)
self.register_button.setFont(font)
self.register_button.setObjectName("register_button")
```

We need to set a series of properties such as font name, font size, font thickness, font style (bold or not, italic or not).

The settings for the input box labels are similar.

```
self.username_label = QtWidgets.QLabel(FIFASystem)
self.username_label.setGeometry(QtCore.QRect(350, 30, 121, 151))
font = QtGui.QFont()
font.setFamily("Times New Roman")
font.setPointSize(16)
font.setItalic(True)
self.username_label.setFont(font)
self.username_label.setStyleSheet("color: rgb(255, 255, 255);")
self.username_label.setScaledContents(False)
self.username_label.setObjectName("username_label")
```

For the table elements of the query screen, I have the following settings.

```
self.tableWidget = QtWidgets.QTableWidget(FIFASystem)
self.tableWidget.setGeometry(QtCore.QRect(-5, 470, 1131, 271))
self.tableWidget.setStyleSheet("background-color: rgb(255, 255, 255, 0.6);")
self.tableWidget.setObjectName("tableWidget")
self.tableWidget.setColumnCount(7)
self.tableWidget.setRowCount(25)
self.tableWidget.horizontalHeader().setVisible(False)
self.tableWidget.horizontalHeader().setDefaultSectionSize(158)
self.tableWidget.verticalHeader().setVisible(False)
```

In addition to the basic settings you also need to set the number of rows and columns, as well as the size of the cells, and then I hide the horizontal and vertical table headers, because they can not set the transparency, affecting the appearance.
Each screen also has the following content that is the same.

```
        self.text(FIFASystem)
        QtCore.QMetaObject.connectSlotsByName(FIFASystem)
```

The first sentence references a *text()* function to display the text on each element, while the second sentence declares that the window receives and sends signals to the corresponding slot function under the name FIFA System.

```
    def text(self, FIFASystem):
        """
        Set interface element text information
        """
        _translate = QtCore.QCoreApplication.translate
        FIFASystem.setWindowTitle(_translate("FIFASystem", "FIFA World Cup System"))
        self.register_button.setText(_translate("FIFASystem", "Register"))
        self.userlogin_button.setText(_translate("FIFASystem", "User Login"))
        self.managerlogin_button.setText(_translate("FIFASystem", "Administrator\n Login"))
        self.username_label.setText(_translate("FIFASystem", "Username:"))
        self.password_label.setText(_translate("FIFASystem", "Password:"))
        self.exit_button.setText(_translate("FIFASystem", "Exit"))
```

The translate class method is used here for text display. This is mainly because it is convenient to manipulate the text in a multilingual scenario (just modify the second parameter directly), where FIFA System is the name of the window object.

Notice that in each screen there is the following line of code.

```
import Source_rc    # Load image
```

In this *Source_rc.py* file we can see that it defines four variables.

```
qt_resource_data = b"\
\x..\x..\x..\...\
"

qt_resource_name = b"\
\x..\x..\x..\...\
"

qt_resource_struct_v1 = b"\
\x..\x..\x..\...\
"

qt_resource_struct_v2 = b"\
\x..\x..\x..\...\
"
```

where *b""* means the string is of type byte, and it should be noted that the default string type in Python3 is utf-8 and *\xss* means the hexadecimal character of *ss*. It can be seen that the content and name of the background image from all walks of life have been converted to hexadecimal byte type strings to facilitate transmission. The main operations used for transmission are as follows.

```
from PyQt5 import QtCore


qt_version = [int(v) for v in QtCore.qVersion().split('.')]
if qt_version < [5, 8, 0]:
    rcc_version = 1
    qt_resource_struct = qt_resource_struct_v1
else:
    rcc_version = 2
    qt_resource_struct = qt_resource_struct_v2

def qInitResources():
    QtCore.qRegisterResourceData(rcc_version, qt_resource_struct, qt_resource_name,
qt_resource_data)

def qCleanupResources():
    QtCore.qUnregisterResourceData(rcc_version, qt_resource_struct, qt_resource_name,
qt_resource_data)

qInitResources()
```

First, choose a different transfer structure according to the version of Qt, and then use the *qInitResources()* function to perform the transfer, which finally enables the background image to be displayed in each sector. With this operation we don't need to put the original png photo file into the project directory, otherwise importing *.png* files would be a complicated task.

Functional Introduction

Page Jumping

Nothing is more important for a graphical interface than interface jumps. A good interface jump is what gives the user a quality experience. So, I will first introduce the page jump function.

Take the login screen jumping to the registration screen as an example.

```
        self.register_button.clicked.connect(self.close)
        self.register_button.clicked.connect(self.register_interface)    # Signal for jumping to the registration screen

    def register_interface(self):
        """
        Slot function for jumping to the registration screen
        """

        from register_interface import RegisterInterface
        self.register_interface = RegisterInterface()
        self.register_interface.show()
```

The parameters in the *clicked.connect()* method here are the logical actions to be performed after the click. Before jumping to the registration screen, the first thing we need to do is to close the original window, then introduce the registration interface file, bind its object instantiation to the class property, and finally use the *show()* function.

Here we need to note that, as mentioned earlier, only window classes that inherit from QMainWindow can be actually ended processes without taking resources in the background, so windows of other interfaces must also inherit from QMainWindow, and their layout class methods (setupUi()) must also be bound in the class properties.

Network Connection and Data Transmission

These are the three main functions on the server side.

```
import socket


def connect_client():
    # get the hostname
    host = socket.gethostname()
```

```
    port = 5000    # initiate port no above 1024

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)    # get instance
    # look closely. The bind() function takes tuple as argument
    server_socket.bind((host, port))    # bind host address and port together
    # configure how many client the server can listen simultaneously
    server_socket.listen(5)
    connection, address = server_socket.accept()    # accept new connection
    return connection, address


def send_message(connection, message):
    connection.send(message.encode('utf-8'))


def receive_response(connection):
    # receive data stream. it won't accept data packet greater than 1024 bytes
    data = connection.recv(1024).decode('utf-8')
    return data
```

First, in the *connect_client()* function, since it is a host-based connection, the host name and port value are obtained first. *server_socket* determines the type of the socket as a network-based (*AF_INET*) TCP socket (*SOCK_STREAM*). Then bind the host name and port value, and finally configure the maximum number of connections while receiving connections from the client.

The *send_message()* function sends the message to be transmitted to the client in utf-8 encoding, and the *receive_response()* function decodes the received message in utf-8. Note that in the data transfer here, the data type can only be a string, so some operations on the string are needed later to get the information we need. (If Chinese is transmitted, gbk should be used for encoding and decoding)

For the client, all interface elements are displayed and user input is in one thread. If the network communication takes too long, the thread will get stuck, which is reflected in the Windows operating system as a program not responding, to avoid this, we put this part in another thread.

```
from PyQt5.QtCore import QThread, pyqtSignal
import socket    # Network Communications # Data Transmission


class TCPClient(QThread):
    """
    Network Communication and Data Transmission
    """
```

```
    display_signal = pyqtSignal(str)

    def __init__(self, data_list):
        super().__init__()
        self.data_list = data_list

    @staticmethod
    def connect_server():
        host = socket.gethostname()    # as both code is running on same pc
        port = 5000    # socket server port number

        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)    # instantiate
        client_socket.connect((host, port))    # connect to the server
        return client_socket

    @staticmethod
    def send_message(client_socket, message):
        client_socket.send(message.encode('utf-8'))

    @staticmethod
    def receive_response(client_socket):
        data = client_socket.recv(1024).decode('utf-8')
        return data

    @staticmethod
    def close_client(client_socket):
        client_socket.shutdown(socket.SHUT_RDWR)
        client_socket.close()

    def run(self):
        client_socket = self.connect_server()
        self.send_message(client_socket, self.data_list)
        response = self.receive_response(client_socket)
        self.display_signal.emit(response)
```

First, we bind the external parameters *data_list* to the class property, then we use static methods to make it easier to receive external parameters for the *connect_server()*, *send_message()*, *receive_response()*, and *close_client()* functions. In the *run()* function for the specific operation of the thread, you can see that it performs an operation to send out data and then receive feedback. At the same time, we define a *display_signal* variable as a signal to the end of the thread to send to the main thread, the signal is a string type, use emit to send the signal out, the content of the signal is the content of the feedback from the server.

For the network part of the call, the server-side operation is

```
import os
from threading import Thread


def main():
    filename = os.path.join(os.path.dirname(__file__), "Data.db")
    clients = []
    while True:
        connection, address = connect_client()
        clients.append(connection)
        Thread(target=get_message, args=(connection, filename)).start()
```

This is the main server-side function, where the path to the database storage is first read, and then connections from clients are received in a loop, while a new thread is opened for each client to use. *get_message()* involves data reading and database operations, which will be explained in detail later.

The following are the operations of network communication in the client.

```
data_list = f'l {username} {password} Administrator'

from client import TCPClient
self.login_thread = TCPClient(data_list)
self.login_thread.display_signal.connect(self.administrator_login_response)
self.login_thread.start()
```

Take the login screen as an example, pass a string parameter, and the signal returned after communicating with the network in a new thread is passed as a parameter to the next function, where *administrator_login_response()* is a class method in the login screen class.

Data Reconfiguration and Storage

Since the type of data to be transferred using sockets for network communication can only be strings, I add symbols to the different kinds of information to distinguish them from other kinds of information when transferring, and when reading such data, we need to do the data reconstruction work accordingly to make sure they are converted into the information we need. The following are some of the styles of data sent by the client.

```
# Login Data
            data_list = f'l {username} {password} User'
            data_list = f'l {username} {password} Administrator'
# Register Data
            data_list = f'r {username} {password} {user_type}'
# Query Data
```

```
                data_list =
f'q|time_type_date_score_team|{time_input}_{type_input}_{date_input}_{score_input}_{team_in
put}'
# Insert Data
                data_list =
f'i_{team_name}_{race_type}_{score}_{date}_{race_time}_{event}_{status}'
# Update Data
                data_list =
f'u_{team_name}_{race_type}_{score}_{date}_{race_time}_{event}_{status}'
```

These data are reconstructed on the server side as follows.

```
def get_message(connection, filename):
    database = connect_database(filename)
    while True:
        function_map = receive_response(connection)

        if function_map == '':
            break
        elif function_map[0] == 'r':
            data_list = function_map.split(' ')
            register_database(connection, database, data_list)
        elif function_map[0] == 'q':
            data = function_map.split('|')
            data_message = data[1].split('_')
            data_list = data[2].split('_')
            query_database(connection, database, data_message, data_list)
        elif function_map[0] == 'l':
            data_list = function_map.split(' ')
            login_database(connection, database, data_list)
        elif function_map[0] == 'i':
            data_list = function_map.split('_')
            insert_database(connection, database, data_list)
        elif function_map[0] == 'u':
            data_list = function_map.split('_')
            update_database(connection, database, data_list)
```

As you can see, the string method *split()* function is used here to convert the original data into a list by separating it with the characters _, |, and space. The server side sends out data in the following style.

```
                database_message = f'{True} {None}'
                database_message = f'{False} {error}'

                database_message = 'False Cannot register duplicate username!'
```

```
database_message = 'False Cannot update duplicate race!'
database_message = 'False Cannot add duplicate race!'
```

On the client side, the first two data styles are handled for

```
response = response.split(' ')
```

For the last three, in addition to the above processing methods, it is necessary to add

```
message = ''
for word in range(1, len(response)):
    message += response[word] + ' '
```

For the queried tournament information, it is a tricky problem to display them in an orderly manner in a table in the GUI, for which the server-side considerations are

```
fields = cursor.fetchall()

reconstitution_data(connection, fields)
```

```
def reconstitution_data(connection, data):
    """
    The format of data queried from database is like [(team name, race type, score, data, time, event, status), (...).]
    """
    if len(data) == 0:
        send_message(connection, f'{False}|Cannot query any information to meet the conditions!')
    else:
        elements = ''
        for row in data:
            for column in row:
                elements += column + '_'
            elements = elements.strip('_') + '&'
        else:
            elements = elements.strip('&')
            send_message(connection, f'{True}|{elements}')
```

After querying all the database information that matches the requirements using the cursor, we get a list of the values of the variablefields. If there is no information that matches the requirements, we get an empty list. We will execute the first part of the condition statement of the *reconstitution_data()* function, i.e. the server returns an error message. If the query finds the required information, the final *elements* variable sent out by the server takes the form of

```
f'{team name1}_{race type1}_{score1}_{data_time1}_{event1}_{status1}&{team name2}_{race type2}_{score2}_{data_time2}_{event2}_{status2}&{team name3}_{race type3}_{score3}_{data_time3}_{event3}_{status3}&···&{team namen}_{race typen}_{scoren}_{data_timen}_{eventn}_{statusn}'
```

After the client receives it, it needs to do the processing, which is done as follows.

```
    def query_response(self, response):
        response = response.split('|')
        if response[0] == 'False':
            """
            If the server cannot receive the message, then it will return a list like 'False|error message'
            """
            try:
                QMessageBox.information(self, 'Error', response[-1])
            except Exception as error:
                print('Error %s' % error)
        else:
            rows = self.tableWidget.rowCount()
            columns = self.tableWidget.columnCount()
            for row in range(rows):
                for column in range(columns):
                    self.tableWidget.setItem(row, column, QtWidgets.QTableWidgetItem(''))
            data = response[-1]
            data_list = data.split('&')
            x = 0
            for row in data_list:
                row_data = row.split('_')
                y = 0
                for column in row_data:
                    self.tableWidget.setItem(x, y, QtWidgets.QTableWidgetItem(str(column)))
                    y += 1
                x += 1
```

The incoming string is first separated by | and if the resulting list has the element False, an exception is thrown. A message box is set up here, and the operation is set up as follows.

```
from PyQt5.QtWidgets import QMessageBox
QMessageBox.information(self, 'Error', response[-1])
```

The *QMessage* module is used to generate the message box, *information()* is the type of the message box, the method has three parameters, the second parameter is the title of the message box, the third parameter is the content of the message box.

If the obtained list does not have the element False, then the else part of the operation is performed. First get the number of rows and columns of the table, and then assign the empty string on each cell (this is to take into account the second query if the number of information is less than the first query, the later cells cannot empty the previous data and thus cause misinterpretation to the user). Then get the query data, the data will be separated by & first, so that the list obtained after the operation like this.

```
[f'{team name1}_{race type1}_{score1}_{data_time1}_{event1}_{status1}', f'{team name2}_{race type2}_{score2}_{data_time2}_{event2}_{status2}', f'{team name3}_{race type3}_{score3}_{data_time3}_{event3}_{status3}',···, f'{team namen}_{race typen}_{scoren}_{data_timen}_{eventn}_{statusn}']
```

Iterating over each element of the resulting list and splitting it according to _ conformity, the resulting list ends up as

```
[f'{team name1}, f'{race type1}', f'{score1}', f'{data_time1}', f'{event1}', f'{status1}']
[f'{team name2}, f'{race type2}', f'{score2}', f'{data_time2}', f'{event2}', f'{status2}']
[f'{team name3}, f'{race type3}', f'{score3}', f'{data_time3}', f'{event3}', f'{status3}']
...
[f'{team namen}, f'{race typen}', f'{scoren}', f'{data_timen}', f'{eventn}', f'{statusn}']
```

the final elements of each list can be displayed on the graphical interface.

For data storage, it is natural to use a database. Here I chose SQLite3, a lightweight database for storage, which has the advantage of not requiring the installation of third-party libraries. There are two tables in this database, which are used to store user data and tournament data respectively. The functions to create the database are as follows.

```
def connect_database(filename):
    create = not os.path.exists(filename)
    database = sqlite3.connect(filename, check_same_thread=False)
    if create:
        cursor = database.cursor()
        cursor.execute("""CREATE TABLE User(
                        id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT
NULL,
                        username TEXT NOT NULL,
                        password TEXT NOT NULL,
                        user_type TEXT NOT NULL );""")

        cursor.execute("""CREATE TABLE Tournament(
                        team_name TEXT NOT NULL,
                        race_type TEXT NOT NULL,
                        score TEXT,
```

```
                    date TEXT NOT NULL,
                    time TEXT NOT NULL,
                    event TEXT,
                    status TEXT NOT NULL);""")
        database.commit()
        initialization_data(database)
        return database
    else:
        return database
```

The first table stores user information with three columns: user name, password, and user type; the second table stores tournament information with seven columns: two team names, tournament type, two team scores, tournament date, tournament time, key events (such as penalty kick scores), and tournament status. The next function, *initialization_data()*, initializes the data in the table where the tournament information is stored. The main content of this function is as follows.

```
def initialization_data(database):
    cursor = database.cursor()
    insert_data = "INSERT INTO Tournament VALUES (?, ?, ?, ?, ?, ?, ?)"
    data = [('Qatar vs Ecuador', 'Group A', '0 · 2', '21 Nov 2022', '00:00', '', 'FT'),
            ('England vs Iran', 'Group B', '6 · 2', '21 Nov 2022', '21:00', '', 'FT'),
            ('Senegal vs Netherlands', 'Group A', '0 · 2', '22 Nov 2022', '00:00', '', 'FT'),
            ('United States vs Wales', 'Group B', '1 · 1', '22 Nov 2022', '03:00', '', 'FT'),
            ('Argentina vs Saudi Arabia', 'Group C', '1 · 2', '22 Nov 2022', '18:00', '', 'FT'),
            ('Denmark vs Tunisia', 'Group D', '0 · 0', '22 Nov 2022', '21:00', '', 'FT'),
            ('Mexico vs Poland', 'Group C', '0 · 0', '23 Nov 2022', '00:00', '', 'FT'),
            ('France vs Australia', 'Group D', '4 · 1', '23 Nov 2022', '03:00', '', 'FT'),
            ('Morocco vs Croatia', 'Group F', '0 · 0', '23 Nov 2022', '18:00', '', 'FT'),
            ('Germany vs Japan', 'Group E', '1 · 2', '23 Nov 2022', '21:00', '', 'FT'),
            ('Spain vs Costa Rica', 'Group E', '7 · 0', '24 Nov 2022', '00:00', '', 'FT'),
            ('Belgium vs Canada', 'Group F', '1 · 0', '24 Nov 2022', '03:00', '', 'FT'),
            ('Switzerland vs Cameroon', 'Group G', '1 · 0', '24 Nov 2022', '18:00', '', 'FT'),
            ('Uruguay vs Korea Republic', 'Group H', '0 · 0', '24 Nov 2022', '21:00', '', 'FT'),
            ('Portugal vs Ghana', 'Group H', '3 · 2', '25 Nov 2022', '00:00', '', 'FT'),
            ('Brazil vs Serbia', 'Group G', '2 · 0', '25 Nov 2022', '03:00', '', 'FT'),
            ('Wales vs Iran', 'Group B', '0 · 2', '25 Nov 2022', '18:00', '', 'FT'),
            ('Qatar vs Senegal', 'Group A', '1 · 3', '25 Nov 2022', '21:00', '', 'FT'),
            ('Netherlands vs Ecuador', 'Group A', '1 · 1', '26 Nov 2022', '00:00', '', 'FT'),
            ('England vs United States', 'Group B', '0 · 0', '26 Nov 2022', '03:00', '', 'FT'),
            ('Tunisia vs Australia', 'Group D', '0 · 1', '26 Nov 2022', '18:00', '', 'FT'),
            ('Poland vs Saudi Arabia', 'Group C', '2 · 0', '26 Nov 2022', '21:00', '', 'FT'),
            ('France vs Denmark', 'Group D', '2 · 1', '27 Nov 2022', '00:00', '', 'FT'),
            ('Argentina vs Mexico', 'Group C', '2 · 0', '27 Nov 2022', '03:00', '', 'FT'),
            ('Japan vs Costa Rica', 'Group E', '0 · 1', '27 Nov 2022', '18:00', '', 'FT'),
```

```
('Belgium vs Morocco', 'Group F', '0 · 2', '27 Nov 2022', '21:00', '', 'FT'),
('Croatia vs Canada', 'Group F', '4 · 1', '28 Nov 2022', '00:00', '', 'FT'),
('Spain vs Germany', 'Group E', '1 · 1', '28 Nov 2022', '03:00', '', 'FT'),
('Cameroon vs Serbia', 'Group G', '3 · 3', '28 Nov 2022', '18:00', '', 'FT'),
('Korea Republic vs Ghana', 'Group H', '2 · 3', '28 Nov 2022', '21:00', '', 'FT'),
('Brazil vs Switzerland', 'Group G', '1 · 0', '29 Nov 2022', '00:00', '', 'FT'),
('Portugal vs Uruguay', 'Group H', '2 · 0', '29 Nov 2022', '03:00', '', 'FT'),
('Netherlands vs Qatar', 'Group A', '2 · 0', '29 Nov 2022', '23:00', '', 'FT'),
('Ecuador vs Senegal', 'Group A', '1 · 2', '29 Nov 2022', '23:00', '', 'FT'),
('Wales vs England', 'Group B', '0 · 3', '30 Nov 2022', '03:00', '', 'FT'),
('Iran vs United States', 'Group B', '0 · 1', '30 Nov 2022', '03:00', '', 'FT'),
('Australia vs Denmark', 'Group D', '1 · 0', '30 Nov 2022', '23:00', '', 'FT'),
('Tunisia vs France', 'Group D', '1 · 0', '30 Nov 2022', '23:00', '', 'FT'),
('Poland vs Argentina', 'Group C', '0 · 2', '1 Dec 2022', '03:00', '', 'FT'),
('Saudi Arabia vs Mexico', 'Group C', '1 · 2', '1 Dec 2022', '03:00', '', 'FT'),
('Croatia vs Belgium', 'Group F', '0 · 0', '1 Dec 2022', '23:00', '', 'FT'),
('Canada vs Morocco', 'Group F', '1 · 2', '1 Dec 2022', '23:00', '', 'FT'),
('Japan vs Spain', 'Group E', '2 · 1', '2 Dec 2022', '03:00', '', 'FT'),
('Costa Rica vs Germany', 'Group E', '2 · 4', '2 Dec 2022', '03:00', '', 'FT'),
('Ghana vs Uruguay', 'Group H', '0 · 2', '2 Dec 2022', '23:00', '', 'FT'),
('Korea Republic vs Portugal', 'Group H', '2 · 1', '2 Dec 2022', '23:00', '', 'FT'),
('Serbia vs Switzerland', 'Group G', '2 · 3', '3 Dec 2022', '03:00', '', 'FT'),
('Cameroon vs Brazil', 'Group G', '1 · 0', '3 Dec 2022', '03:00', '', 'FT'),
('Netherlands vs United States', 'Round of 16', '3 · 1', '3 Dec 2022', '23:00', '', 'FT'),
('Argentina vs Australia', 'Round of 16', '2 · 1', '4 Dec 2022', '03:00', '', 'FT'),
('France vs Poland', 'Round of 16', '3 · 1', '4 Dec 2022', '23:00', '', 'FT'),
('England vs Senegal', 'Round of 16', '3 · 0', '5 Dec 2022', '03:00', '', 'FT'),
('Japan vs Croatia', 'Round of 16', '1 · 1 (1) · (3)', '5 Dec 2022', '23:00',
  'Croatia wins 3 - 1 on penalties', 'FT'),
('Brazil vs Korea Republic', 'Round of 16', '4 · 1', '6 Dec 2022', '03:00', '', 'FT'),
('Morocco vs Spain', 'Round of 16', '0 · 0 (3) · (0)', '6 Dec 2022', '23:00',
  'Morocco wins 3 - 0 on penalties', 'FT'),
('Portugal vs Switzerland', 'Round of 16', '6 · 1', '7 Dec 2022', '03:00', '', 'FT'),
('Croatia vs Brazil', 'Quarter-final', '1 · 1 (4) · (2)', '9 Dec 2022', '23:00',
  'Croatia wins 4 - 2 on penalties', 'FT'),
('Netherlands vs Argentina', 'Quarter-final', '2 · 2 (3) · (4)', '10 Dec 2022', '03:00',
  'Argentina wins 4 - 3 on penalties', 'FT'),
('Morocco vs Portugal', 'Quarter-final', '1 · 0', '10 Dec 2022', '23:00', '', 'FT'),
('England vs France', 'Quarter-final', '1 · 2', '11 Dec 2022', '03:00', '', 'FT'),
('Argentina vs Croatia', 'Semi-final', '3 · 0', '14 Dec 2022', '03:00', '', 'FT'),
('France vs Morocco', 'Semi-final', '2 · 0', '15 Dec 2022', '03:00', '', 'FT'),
('Croatia vs Morocco', 'Play-off for third place', '2 · 1', '17 Dec 2022', '23:00', ''
'FT'),
('Argentina vs France', 'Final', '3 · 3 (4) · (2)', '18 Dec 2022', '23:00',
```

```
                'Argentina wins 4 - 2 on penalties', 'FT')]
    cursor.executemany(insert_data, data)
    database.commit()
```

We can use regular expressions to represent a common format for this data.

```
team_name = ([A-Z][a-z]+( [A-Z][a-z]+)* vs [A-Z][a-z]+( [A-Z][a-z]+)*)
race_type = (Group [A-H]|Round of 16|Quarter-final|Semi-final|Play-off for third place|Final)
score = (\d · \d( [(]\d[)] · [(]\d[)])?)
date = (\d{1,2} [A-Z][a-z]{2} [0-9]{4})
time = (\d{2}:\d{2})
status = (FT|HT|I|A|ET|NS)
```

In the tournament status, we specify FT for full time (i.e., game completed), HT for half time (i.e., first half of the game completed), I for intermission, A for penalty, ET for overtime, and NT for game not started. When using the GUI for input, we take the following regular expressions to prevent interfering characters before and after the characters.

For characters such as date, time and score, which are mainly composed of numbers, our solution is

```
^(.\D+)*(regex)(.+)*$
```

For team name, race type, and status, which are mainly alphabetic characters, our scheme is

```
^(.\d+)*(regex)(.+)*$
```

We may enter multiple strings that match the date, time, team name, and score, and here we specify to match the last string that matches, so the final regular expression is

```
import re


re.compile(r'^(.\D+)*(regex)+(.+)*$')
re.compile(r'^(.\d+)*(regex)+(.+)*$')
```

Of course, our regular expressions will be a little different when the user makes a tournament query, because the query is a fuzzy query and we need to reflect the most comprehensive information that the user wants to know as much as possible. This part will be described in detail later.

Web Crawler

The data obtained by the web crawler is the data inserted into the database using the initialization_data() function. Because the crawler operation requires high environmental configuration, it is not added to the server file. The operation of the web crawler is in *web_crawler.py*, which is related as follows.

```
import pyautogui
```

```
import time
import re
from selenium.webdriver import Edge

web = Edge()
url = 'https://www.fifa.com/fifaplus/en/tournaments/mens/worldcup/qatar2022/scores-fixtures?country=CN&wtw-filter=ALL'
web.get(url)
```

First start the browser driver and connect to the appropriate URL.

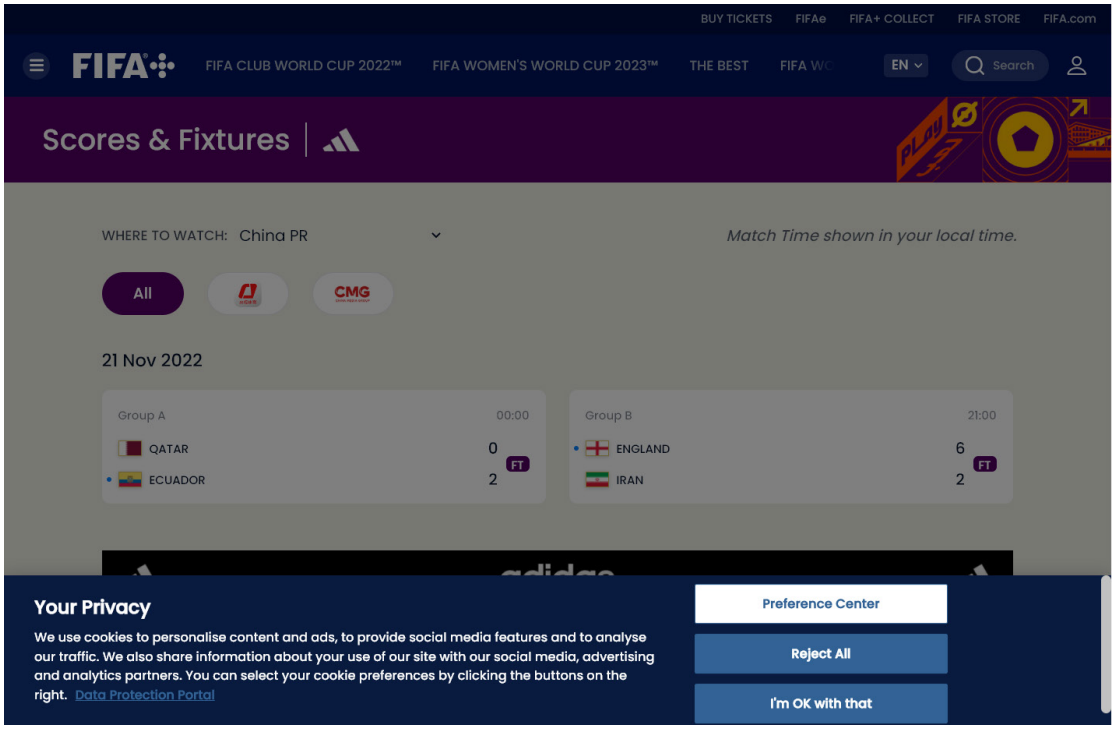


Figure 4: Initial website interface

```
pyautogui.moveTo(937, 1020, duration=3)
time.sleep(4)
pyautogui.click()
text = web.page_source
```

Notice that the site opens with a pop-up for the privacy policy, which we need to close to see the specific tournament data. So, using the *moveTo* method of the pyautogui module will take 3 seconds for the cursor to move to the specified location and then the program sleeps for 4 seconds to simulate user reaction time. Finally get the source code of the site after the JavaScript rendering is done. In fact, it took 7 seconds for the page to complete the corresponding rendering. After practice, we found that we could skip the step of simulating the cursor movement by clicking on it, because the privacy policy of this page and the tournament data are in the same JavaScript file, so this method above is very useful for pages that need to click on the privacy policy before they can fully display the data behind it. Next, use regular expressions to query for data that matches the requirements and

store it.

```
date = re.findall(r'<div class="matches-container_title__1uTPf">(.*?)</div>', text)
race_time = re.findall(r'<div class="match-block_wtwStadiumName__2EACw">(.*?)</div>',
text)
race_type = re.findall(r'<div class="match-block_wtwStadiumName__2EACw ff-mb-
0">(.*?)</div>', text)
team_name = re.findall(r'<div class="wtw-teams-horizontally-component_TeamName__2lZ2s ff-
mb-0 ff-ml-4">(.*?)</div>',
                       text)
score_first = re.findall(r'<div class="wtw-teams-horizontally-
component_score1__3HTmk">(.*?)</div>', text)
score_second = re.findall(r'<div class="wtw-teams-horizontally-
component_score2__20sPm">(.*?)</div>', text)
status = re.findall(r'<div class="wtw-match-status_indicator__3cO5Z wtw-match-
status_fullTime__1ONKu">(.*?)</div>',
                    text)
penalties_score_first = re.findall(r'<div class="wtw-teams-horizontally-
component_penaltiesColumn1__tC-Ud">(.*?)</div>',
                                   text)
penalties_score_second = re.findall(
    r'<div class="wtw-teams-horizontally-
component_penaltiesColumn2__sVcHw">(.*?)</div>', text)
event = re.findall(r'<p class="ff-m-0">(.*?)</p>', text)
web.close()
```

We can see that the data we need are stored in the corresponding class of the div tag in the HTML code, so we can use the *findall()* method to get them. The next step is to reintegrate the data. We note that the events do not correspond to the dates one by one, while the names and scores of the two teams are separated for each game, and the games without penalty events are not in the list of events. The data integration is done as follows.

```
team_list = []
for i in range(len(team_name)):
    if i % 2 == 0:
        team_list.append(team_name[i] + ' vs ' + team_name[i + 1])

score_data = []
for i in range(len(score_first)):
    score_data.append(score_first[i] + ' · ' + score_second[i])

penalties_data = []
for i in range(len(penalties_score_first)):
    penalties_data.append(penalties_score_first[i] + ' · ' + penalties_score_second[i])
```

The list of team names is easy to put together. Just let the element with an even index value plus the ' vs ' character be spliced with the latter element. (Index values start from 0)

For the list of scores and penalty scores, simply stitch the scores of the two teams according to the index values, and of course eventually continue to stitch the two lists obtained into one list.

```
score_list = []
event_list = []
n = 0
for i in range(len(race_type)):
    if (race_type[i] == 'Round of 16' or race_type[i] == 'Quarter-final' or race_type[i] == 'Final') and (score_data[i][0] == score_data[i][-1]):
        score_list.append(score_data[i]+' '+penalties_data[n])
        event_list.append(event[n].replace('  ', ' '))
        n += 1
    else:
        score_list.append(score_data[i])
        event_list.append('')
```

This is because penalty events only occur when the penalty conditions are met, and the penalty match cannot be a group match, while the two teams need to have the same score before the penalty. The only matches that meet the conditions in the above data are the round of 16, quarterfinals and finals. So, in this case the two team scores and the penalty scores are separated by spaces and spliced together and added to a new list, with the corresponding event replacing the original ' ' with an empty string. (' ' means space in the HTML code) In the rest of the cases, the empty string is added directly to the new list. The last thing we need to deal with is the date.

```
date_list = []
m = 0
j = 3
k = 2
for i in range(len(race_time)):
    if i in (0, 1):
        date_list.append(date[m])
    elif i in (46, 47, 48):
        k += 1
        if k % 3 == 0:
            m += 1
            date_list.append(date[m])
        else:
            date_list.append(date[m])
    elif i in (49, 50, 51, 52, 53, 54, 57, 58):
        if (i+1) % 2 == 0:
            m += 1
```

```
            date_list.append(date[m])
        else:
            date_list.append(date[m])
    elif i in (55, 56, 59, 60, 61, 62, 63):
        m += 1
        date_list.append(date[m])
    else:
        j += 1
        if j % 4 == 0:
            m += 1
            date_list.append(date[m])
        else:
            date_list.append(date[m])
```

In total, there are 64 games with an index value of 0-63. We note that: the first two matches are played on one day according to Beijing time; matches with index values 55, 56, 59, 60, 61, 62, 63 are the only matches on that day; matches with index values 49, 50, 51, 52, 53, 54, 57, 58 are played on the days when two matches are played; matches with index values 46, 47, 48 are played on the same day. The matches with index values 46, 47, 48 are held on the same day. Finally, the data are combined into a list according to the index value.

```
data = []
for i in range(len(team_list)):
    data.append((team_list[i], race_type[i], score_list[i], date_list[i], race_time[i], event_list[i], status[i]))

print(data)
```

Specific Functions

Login

- Client

```
self.userlogin_button.clicked.connect(self.user_interface)    # Signal for jumping to the user screen

self.managerlogin_button.clicked.connect(self.manager_interface)    # Signal for jumping to the administrator screen
```

The mouse clicks on each of the two buttons to jump to the corresponding function to be executed. The operation for normal user login is

```
def user_interface(self):
        """
```

```
        Slot function for jumping to the user screen
        """
        if self.username_line.text().strip() == '' or self.password_line.text().strip() == '':
            try:
                QMessageBox.information(self, 'Error', 'Invalid input, please try again.')
            except Exception as error:
                print('Input error %s' % error)
        else:
            username = self.username_line.text().strip()
            password = self.password_line.text().strip()

            data_list = f'l {username} {password} User'

            from client import TCPClient
            self.login_thread = TCPClient(data_list)
            self.login_thread.display_signal.connect(self.user_login_response)
            self.login_thread.start()

    def user_login_response(self, response):
        response = response.split(' ')
        if response[0] == 'False':
            """
            If the server cannot receive the message, then it will return a list like 'False error
message'
            """
            try:
                message = ''
                for word in range(1, len(response)):
                    message += response[word] + ' '
                QMessageBox.information(self, 'Error', message)
            except Exception as error:
                print('Error %s' % error)
        else:
            try:
                QMessageBox.information(self, 'Welcome', 'Successful Login')
            except Exception as error:
                print('Database error %s' % error)
            else:
                self.close()
                from user_interface import UserInterface
                self.user_interface = UserInterface()
                self.user_interface.show()
```

Get the entered username and password, the strip method is used to avoid space interference. If no

content is entered, the exception is handled, if there is no exception, the network communication is called and the signal thrown by the communication thread is stored as response and handled accordingly, finally, if there is no exception, the message box of successful login is popped up and the page is jumped. The process for administrator login is the same, with the following changes to the details.

```
data_list = f'l {username} {password} Administrator'
```

```
from administrator_interface import AdministratorInterface
self.administrator_interface = AdministratorInterface()
self.administrator_interface.show()
```

- Server

```
def login_database(connection, database, data_list):
    username = data_list[1]
    password = data_list[2]
    user_type = data_list[3]

    database_message = get_registration(database, username, password, user_type)
    send_message(connection, database_message)


def get_registration(database, username, password, user_type):
    cursor = database.cursor()
    cursor.execute("SELECT * FROM User WHERE username=? AND password=? AND
user_type=?",
                   (username, password, user_type))

    # Check if the user is registered
    fields = cursor.fetchone()  # It will return a tuple like (password, ) if no data, then return
None
    if fields is None:
        database_message = 'False Cannot login unregistered user'
        return database_message
    else:
        database_message = f'{True} {None}'
        return database_message
```

The server side receives the information from the client and reorganizes it. It queries the database whether the input information is registered or not, and returns an error message if it is not registered, otherwise it returns a successful login message.

Registration

- Client

First jump to the registration screen.

```
    def register_interface(self):
        """
        Slot function for jumping to the registration screen
        """

        from register_interface import RegisterInterface
        self.register_interface = RegisterInterface()
        self.register_interface.show()

        self.register_button.clicked.connect(self.after_register)    # Signals for completing
registration

    def after_register(self):
        """
        Slot function for completing registration
        """
        if self.username_line.text().strip() == '' or self.password_line.text().strip() == '' \
                or self.password_line_2.text().strip() == '':
            try:
                QMessageBox.information(self, 'Error', 'Invalid input, please try again.')
            except Exception as error:
                print('Input error %s' % error)
        elif len(self.password_line.text().strip()) < 6:
            QMessageBox.information(self, 'Warning', 'The length of your password is less
than 6.')
        elif self.password_line.text().strip() != self.password_line_2.text().strip():
            try:
                QMessageBox.information(self, 'Error', 'The passwords of input are not the
same.')
            except Exception as error:
                print('Unknown error %s' % error)
        else:
            username = self.username_line.text().strip()
            password = self.password_line.text().strip()
            user_type = self.comboBox.currentText().strip()

            data_list = f'r {username} {password} {user_type}'
            from client import TCPClient

            self.registration_thread = TCPClient(data_list)
```

```
                self.registration_thread.display_signal.connect(self.registration_response)
                self.registration_thread.start()

    def registration_response(self, response):
        response = response.split(' ')
        if response[0] == 'False':
            """
            If the server cannot receive the message, then it will return a list like 'False error message'
            """
            try:
                message = ''
                for word in range(1, len(response)):
                    message += response[word] + ' '
                QMessageBox.information(self, 'Error', message)
            except Exception as error:
                print('Error %s' % error)
        else:
            try:
                reply = QMessageBox.question(self, 'Return', 'Successful Registration. Really return?',
                                             QMessageBox.Yes | QMessageBox.No | QMessageBox.Cancel,
                                             QMessageBox.Cancel)
            except Exception as error:
                print('Database error %s' % error)
            else:
                if reply == QMessageBox.Yes:
                    self.close()
                    from main_interface import UIMainWindow
                    self.main_interface = UIMainWindow()
                    self.main_interface.show()
                else:
                    pass
```

Get the text content of the input box and dropdown box, integrated into a data list. If the password is less than 6 digits, the password entered twice is not the same and no content is entered are handled exceptionally, otherwise the data is sent to the server, the client receives the returned signal and processes it, no exceptions are shown then a dialog box pops up for the user to choose. If the user selects Yes, the user will exit the registration interface and return to the login interface, otherwise the original state will be maintained.

- Server

```
def register_database(connection, database, data_list):
```

```
        username = data_list[1]
        password = data_list[2]
        user_type = data_list[3]

        database_message = new_registration(database, username, password, user_type)
        send_message(connection, database_message)


def new_registration(database, username, password, user_type):
        cursor = database.cursor()
        cursor.execute("SELECT username FROM User WHERE username=? AND user_type=?",
(username, user_type))

        # Check if the username is registered
        fields = cursor.fetchone()    # It will return a tuple like (password, ) if no data, then return
None
        if fields is not None:
                previous_data = fields[0]
                if username == previous_data:
                        database_message = 'False Cannot register duplicate username!'
                        return database_message

                # If the student is not registered, insert the data into the database
        try:
                cursor.execute("INSERT INTO User (username, password, user_type) VALUES
(?, ?, ?)",
                                        (username, password, user_type))
                database.commit()
        except Exception as error:
                database_message = f'{False} {error}'
                return database_message
        else:
                database_message = f'{True} {None}'
                return database_message
```

The server receives the incoming data and queries the database to see if there is the same user name under the same type, if it finds the same user name then the user name has been registered and an error message is returned, otherwise the registration completion message is returned.

Add Tournaments

- Client

```
        self.insertButton.clicked.connect(self.after_insert)    # Signal for completing insert
```

Transmits the operation signal to the corresponding function via the button.

```
    def after_insert(self):
        team_name = self.teamname_line.text().strip()
        race_type_input = self.racetype_line.text()
        score = self.score_line.text().strip()
        date_input = self.date_line.text()
        race_time_input = self.time_line.text()
        event = self.event_line.text().strip()
        status_input = self.status_line.text()

        date_regex = re.compile(r'^(.\D+)*(\d{1,2} [A-Z][a-z]{2} [0-9]{4})+(.+)*$')
        time_regex = re.compile(r'^(.\D+)*(\d{2}:\d{2})+(.+)*$')
        race_type_regex = re.compile(
            r'^(.\d+)*(Group [A-H]|Round of 16|Quarter-final|Semi-final|Play-off for third
place|Final)(.+)*$')
        status_regex = re.compile(r'FT|HT|I|A|ET|NS')

        date_data = re.search(date_regex, date_input)
        race_time_data = re.search(time_regex, race_time_input)
        race_type_data = re.search(race_type_regex, race_type_input)
        status_data = re.search(status_regex, status_input)

        if (date_data is None) or (race_type_data is None) or (race_time_data is None) or
(status_data is None):
            try:
                QMessageBox.information(
                    self, 'Error', 'Invalid input, please input correct format information you
want to insert.')
            except Exception as error:
                print('Input error %s' % error)
        else:
            date = date_data.group(2)
            race_time = race_time_data.group(2)
            race_type = race_type_data.group(2)
            status = status_data.group()
            data_list =
f'i_{team_name}_{race_type}_{score}_{date}_{race_time}_{event}_{status}'

            from client import TCPClient
            self.insert_thread = TCPClient(data_list)
            self.insert_thread.display_signal.connect(self.insert_response)
            self.insert_thread.start()
```

```
    def insert_response(self, response):
        response = response.split(' ')
        if response[0] == 'False':
            """
            If the server cannot receive the message, then it will return a list like 'False error
message'
            """
            try:
                message = ''
                for word in range(1, len(response)):
                    message += response[word] + ' '
                QMessageBox.information(self, 'Error', message)
            except Exception as error:
                print('Error %s' % error)
        else:
            try:
                QMessageBox.information(self, 'Insertion', 'Successful Insertion.')
            except Exception as error:
                print('Error %s' % error)
```

The operation after receiving the server signal is basically the same as the previous one, so we will not repeat it here. We specify that adding an event requires at least the date, time, event type and event status of the event. Depending on the game later on, the names of the two teams are updated as well as the score and possible penalty events.

- Server

```
def insert_database(connection, database, data_list):
    team_name = data_list[1]
    race_type = data_list[2]
    score = data_list[3]
    date = data_list[4]
    race_time = data_list[5]
    event = data_list[6]
    status = data_list[7]

    database_message = insert_race(database, team_name, race_type, score, date, race_time,
event, status)
    send_message(connection, database_message)


def insert_race(database, team_name, race_type, score, date, race_time, event, status):
    cursor = database.cursor()
    cursor.execute("SELECT * FROM Tournament WHERE race_type=? AND date=? AND
time=? "
```

```
                    "AND status=?", (race_type, date, race_time, status))
        # Check if the race is inserted
        fields = cursor.fetchone()    # It will return a tuple like (password, ) if no data, then return
None
        if fields is None:
            try:
                cursor.execute("INSERT INTO Tournament VALUES (?, ?, ?, ?, ?, ?, ?)",
                               (team_name, race_type, score, date, race_time, event, status))
                database.commit()
            except Exception as error:
                database_message = f'{False} {error}'
                return database_message
            else:
                database_message = f'{True} {None}'
                return database_message
        else:
            database_message = 'False Cannot add duplicate race!'
            return database_message
```

After the server receives the data from the client, we need to check if there is already another tournament for the same tournament time, tournament date, and tournament status under the same tournament type. If there is already a corresponding race, an exception message is returned, and if there is no registration, this data is inserted into the database.

Update Tournaments

- Client

```
            self.updateButton.clicked.connect(self.after_update)    # Signal for completing upgrade


        def after_update(self):
            ...
            date_regex = re.compile(r'^(.\D+)*(\d{1,2} [A-Z][a-z]{2} [0-9]{4})+(.+)*$')
            time_regex = re.compile(r'^(.\D+)*(\d{2}:\d{2})+(.+)*$')
            team_name_regex = re.compile(r'(.\d+)*([A-Z][a-z]+( [A-Z][a-z]+)* vs [A-Z][a-
z]+( [A-Z][a-z]+)*)+(.+)*$')
            score_regex = re.compile(r'^(.\D+)*(\d · \d( [(]\d[)] · [(]\d[)])?)+(.+)*$')
            race_type_regex = re.compile(
                r'^(.\d+)*(Group [A-H]|Round of 16|Quarter-final|Semi-final|Play-off for third
place|Final)(.+)*$')
            status_regex = re.compile(r'FT|HT|I|A|ET|NS')

            date_data = re.search(date_regex, date_input)
```

```
            race_time_data = re.search(time_regex, race_time_input)
            team_name_data = re.search(team_name_regex, team_name_input)
            score_data = re.search(score_regex, score_input)
            race_type_data = re.search(race_type_regex, race_type_input)
            status_data = re.search(status_regex, status_input)

            if (date_data is None) or (race_type_data is None) or (race_time_data is None) or
(status_data is None)\
                    or (team_name_data is None) or (score_data is None):
                try:
                    QMessageBox.information(
                        self, 'Error', 'Invalid input, please input correct format information you
want to upgrade.')
                except Exception as error:
                    print('Input error %s' % error)
            else:
                date = date_data.group(2)
                race_time = race_time_data.group(2)
                race_type = race_type_data.group(2)
                score = score_data.group(2)
                team_name = team_name_data.group(2)
                status = status_data.group()
                data_list =
f'u_{team_name}_{race_type}_{score}_{date}_{race_time}_{event}_{status}'
```

The operation of updating a tournament on the client side is basically the same as adding a tournament, but the input data requirements are a bit more stringent. When updating data, we require all data except for penalty events to meet the requirements.

- Server

The server-side operation is similar to that of adding a tournament, except that the following parts differ.

```
def update_race(database, team_name, race_type, score, date, race_time, event, status):
    cursor = database.cursor()
    cursor.execute("SELECT * FROM Tournament WHERE team_name=? AND race_type=?
AND score=? AND date=? AND time=? "
                   "AND event=? AND status=?", (team_name, race_type, score, date,
race_time, event, status))
    # Check if the race is inserted
    fields = cursor.fetchone()  # It will return a tuple like (password, ) if no data, then return
None
    if fields is None:
```

```
        try:
            cursor.execute(
                "UPDATE Tournament SET team_name=?, score=?, event=?,
status=?WHERE race_type=? AND date=? AND time=?",
                (team_name, score, event, status, race_type, date, race_time))
            database.commit()
```

Query Tournaments

- Client

As mentioned before, this query is a fuzzy query, so for the game date and team name, our format needs to be set more broadly. The format of these two queries is

```
        date_regex = re.compile(r'^(.\D+)*(\d{1,2} [A-Z][a-z]{2}( [0-9]{4})?)+(.+)*$')
        team_name_regex = re.compile(r'^(.\d+)*([A-Z][a-z]+(( [A-Z][a-z]+)* vs [A-Z][a-
z]+( [A-Z][a-z]+)*)?)+(.+)*$')
```

With this change, the year can be omitted from the input date, and the team name can be entered as just a country name. We stipulate that only the following five types of information can be entered for a query.

```
        team_name = re.search(team_name_regex, team_name_data)
        score = re.search(score_regex, score_data)
        date = re.search(date_regex, date_data)
        race_type = re.search(race_type_regex, race_type_data)
        race_time = re.search(time_regex, time_data)
```

The user may have entered 0, 1, 2, 3, 4 or 5 of these five types of information, so we need to determine their input to send the data. If the user does not enter any of the information, the following operation is performed.

```
        if (team_name is None) and (score is None) and (date is None) and (race_type is None)
and (race_time is None):
            try:
                QMessageBox.information(self, 'Error', 'Invalid input, please input some
information you want to know.')
            except Exception as error:
                print('Input error %s' % error)
```

If the user enters one type of information, there are five cases.

```
        elif (team_name is None) and (score is None) and (date is None) and (race_type is
None):
            time_input = race_time.group(2)

            data_list = f'q|time|{time_input}'
```

```
        elif (team_name is None) and (score is None) and (date is None) and (race_time is
None):
            type_input = race_type.group(2)

            data_list = f'q|type|{type_input}'

        elif (team_name is None) and (score is None) and (race_type is None) and (race_time is
None):
            date_input = date.group(2)

            data_list = f'q|date|{date_input}'

        elif (team_name is None) and (date is None) and (race_type is None) and (race_time is
None):
            score_input = score.group(2)

            data_list = f'q|score|{score_input}'

        elif (score is None) and (date is None) and (race_type is None) and (race_time is
None):
            team_input = team_name.group(2)

            data_list = f'q|team|{team_input}'
```

If the user enters two or three types of information, there are 10 cases respectively, which are not repeated here. If the user enters four kinds of information, there are five cases, and if he enters five kinds of information, there is one case. The final data sending operation is the same.

```
            self.query_thread = TCPClient(data_list)
            self.query_thread.display_signal.connect(self.query_response)
            self.query_thread.start()
```

- Server

The server will query the user according to the type and number of information entered by the user, here we take the case where the user has entered all the information as an example.

```
def query_database(connection, database, data_message, data_list):
    cursor = database.cursor()
    ...
    ...
    ...
    else:
        race_time = data_list[0]
        race_type = data_list[1]
        date = data_list[2]
```

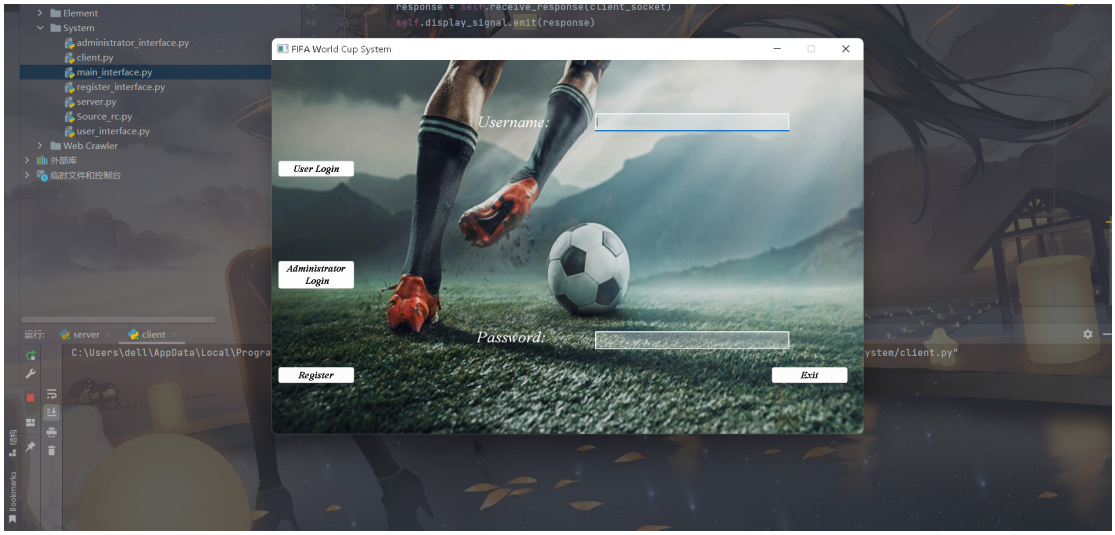
```
            score = data_list[3]
            team_name = data_list[4]
            cursor.execute(
                "SELECT * FROM Tournament WHERE time LIKE ? AND race_type LIKE ? AND
date LIKE ? AND score LIKE ? AND "
                "team_name LIKE ?",
                (f'%{race_time}%', f'%{race_type}%', f'%{date}%', f'%{score}%',
f'%{team_name}%'))
            fields = cursor.fetchall()

            reconstitution_data(connection, fields)
```

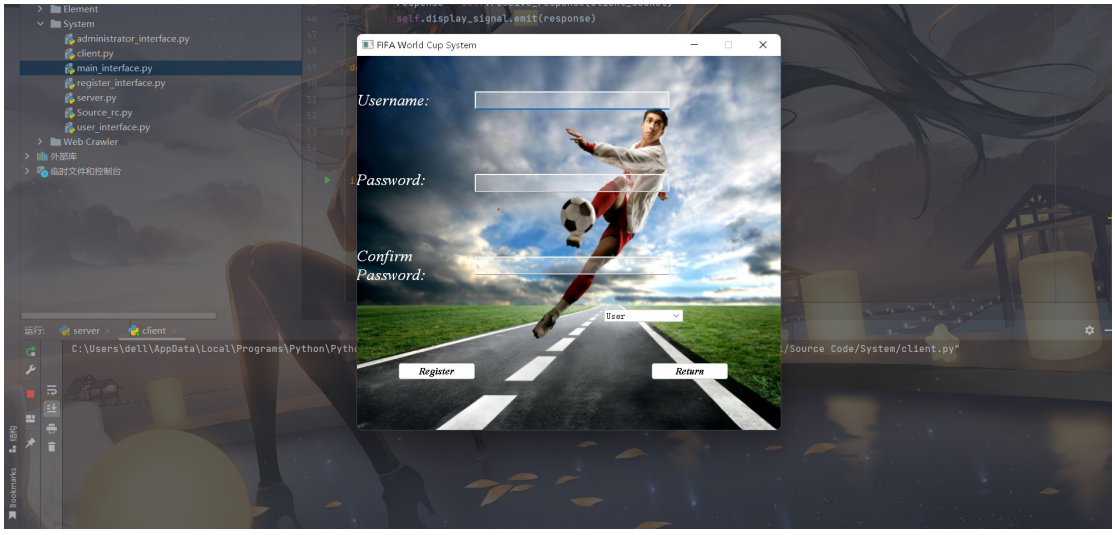
Notice that our database query statement uses the LIKE clause to implement a fuzzy query.

System Demonstration

Run the server.py and client.py files first, and the results are as follows.

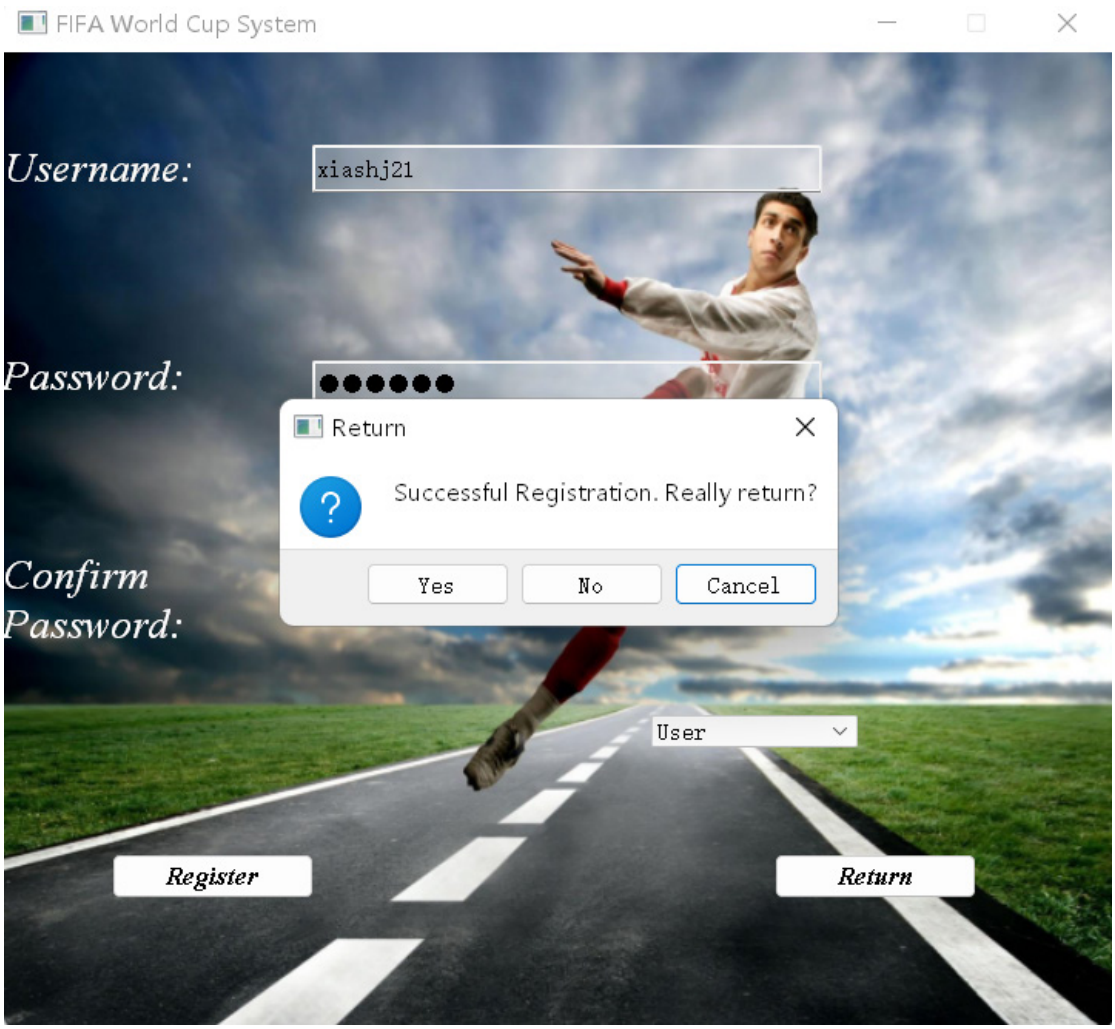


Click the Register button, and the results are as follows.

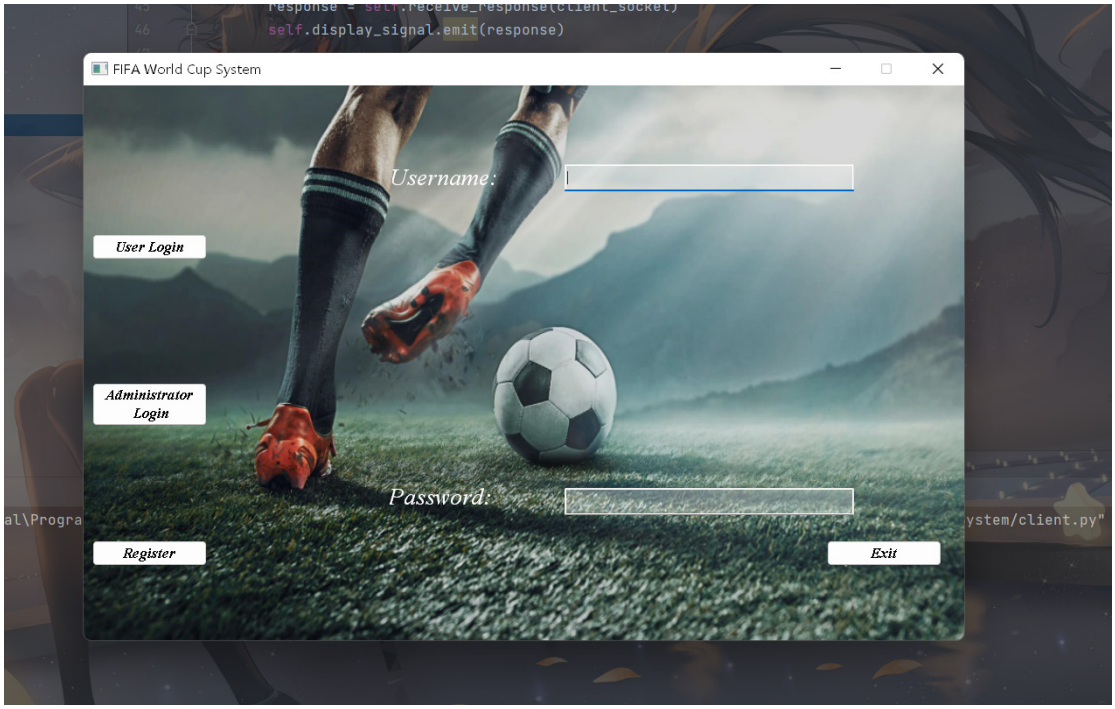


Registered user name xiashj21, password is 123456, user type is normal user.

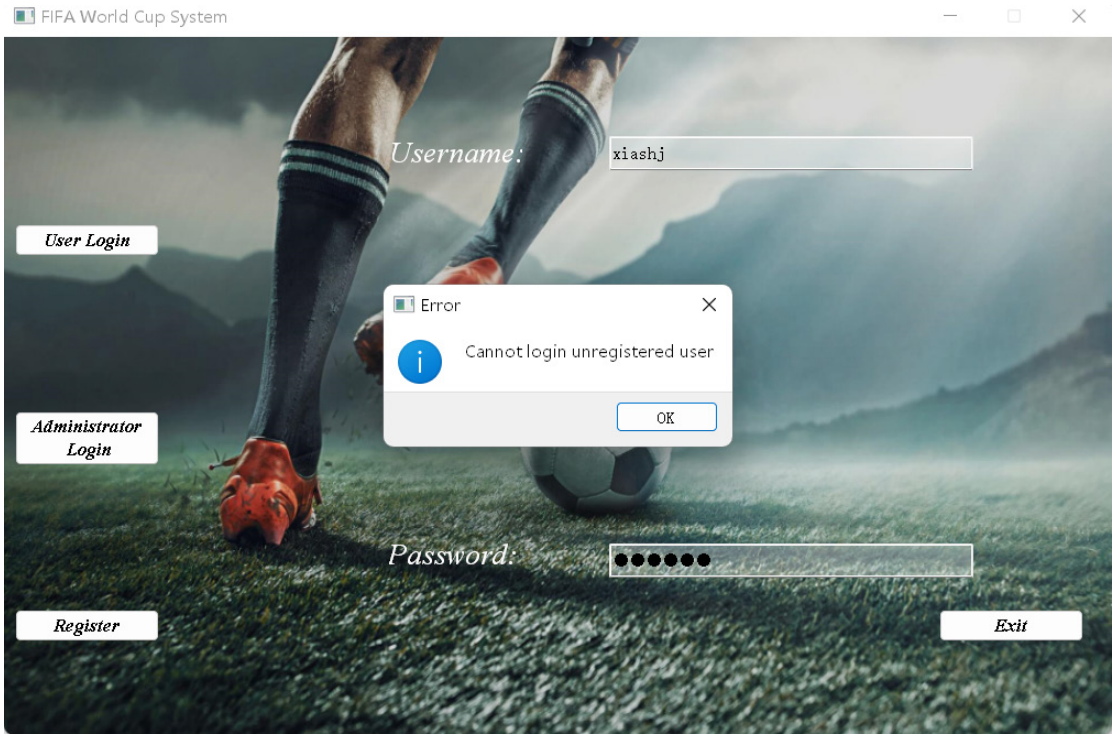
Click the Register button.



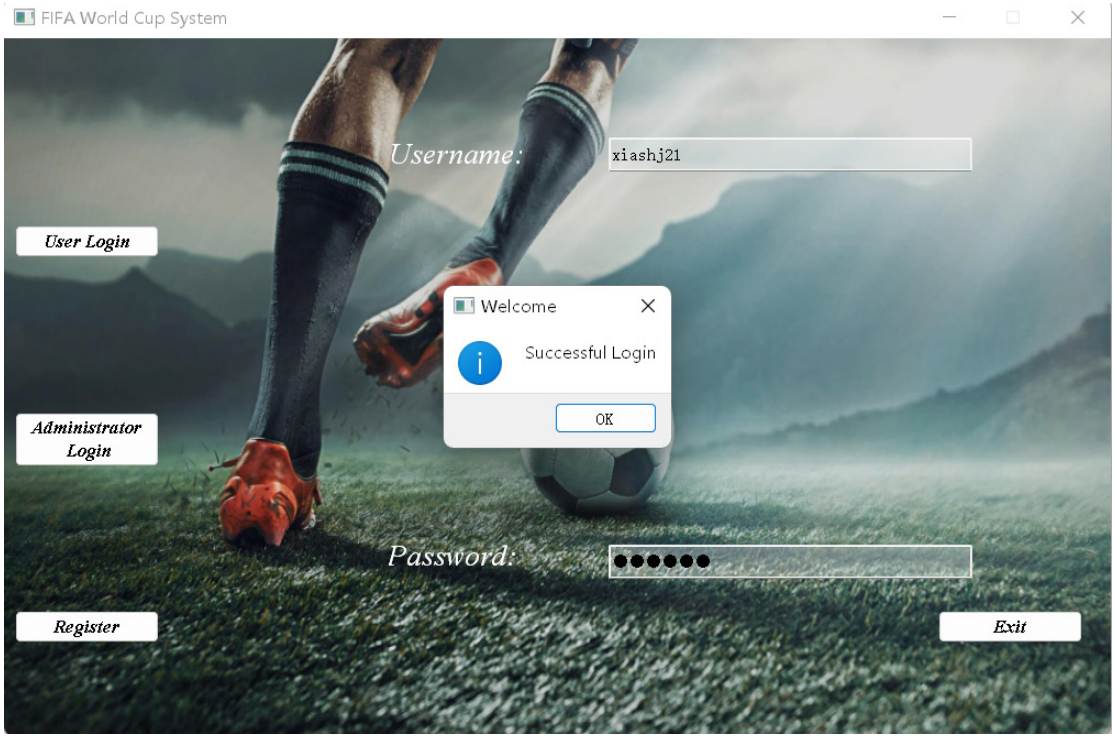
Click Yes, return to the main interface.



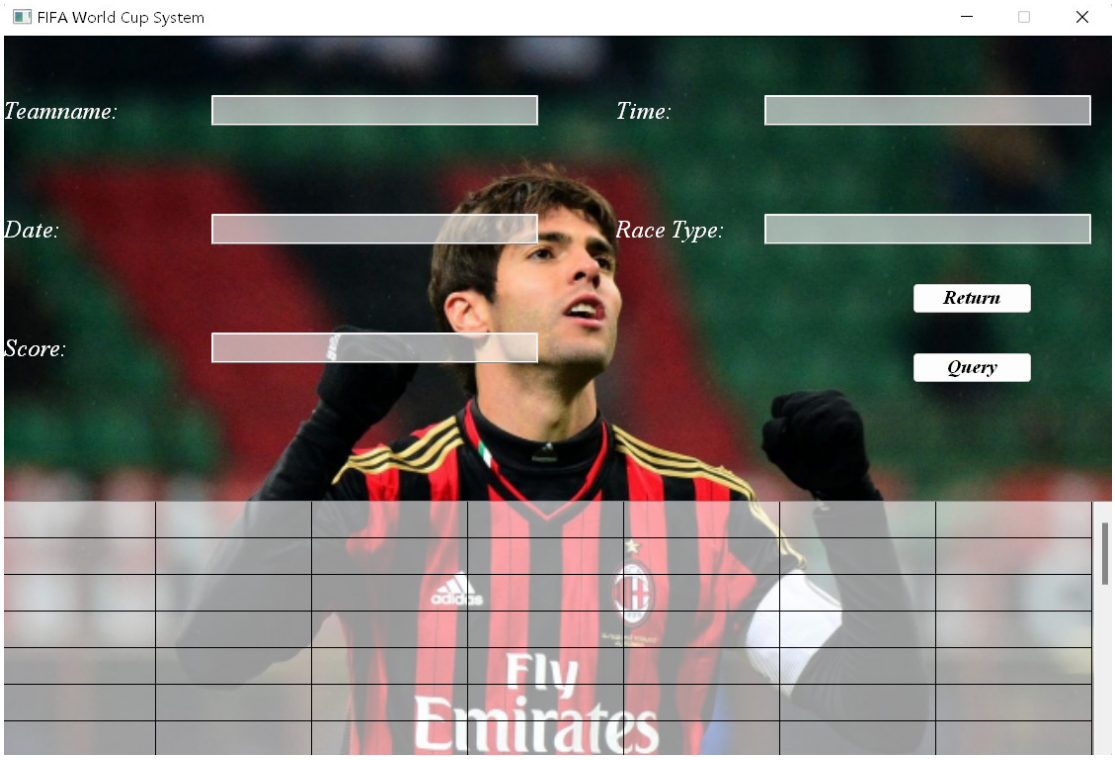
If we enter information other than what we just registered, for example, enter the user name is xiashj, password is 123456, click the User Login button.



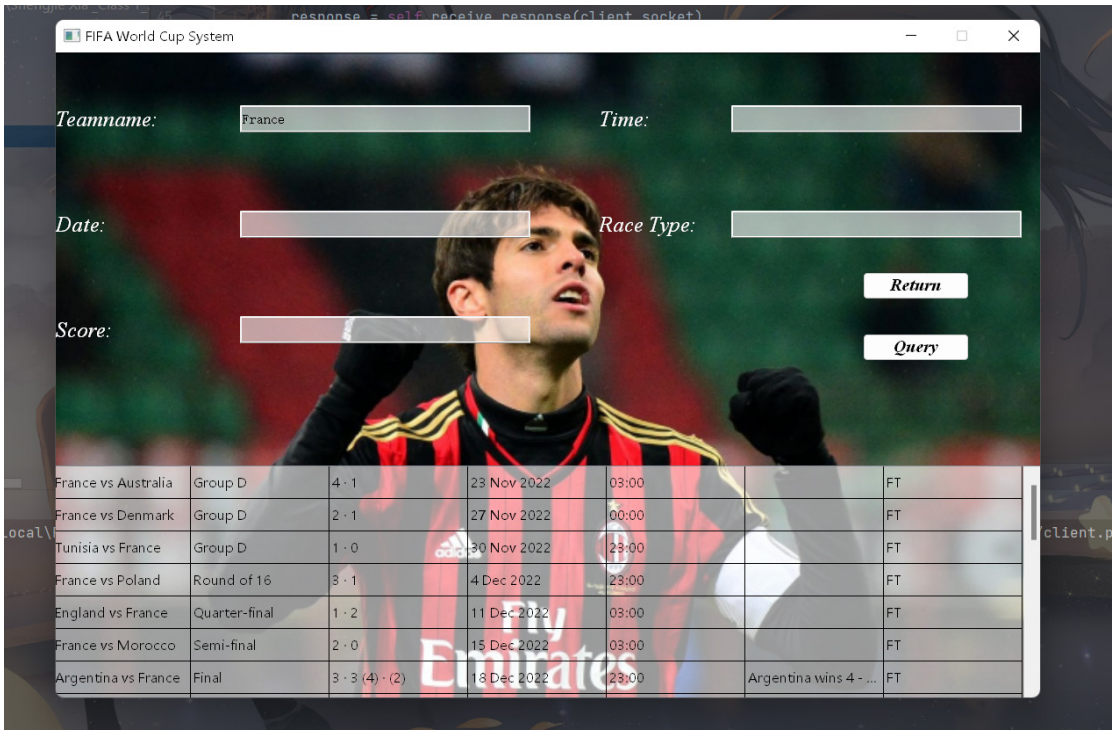
Enter the correct information, it will change to



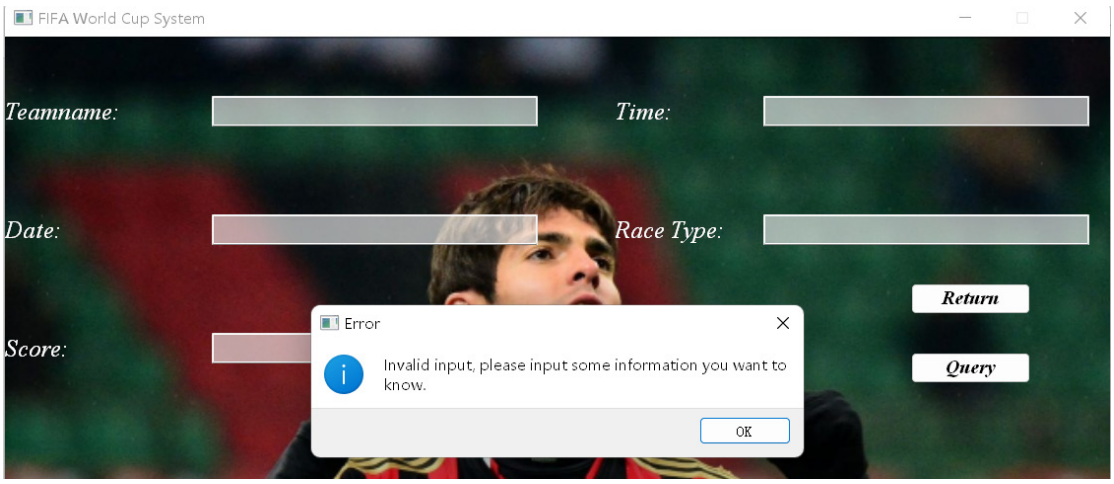
Click the OK button for page jumping.



We enter France in the Teamname box and click on the Query button.

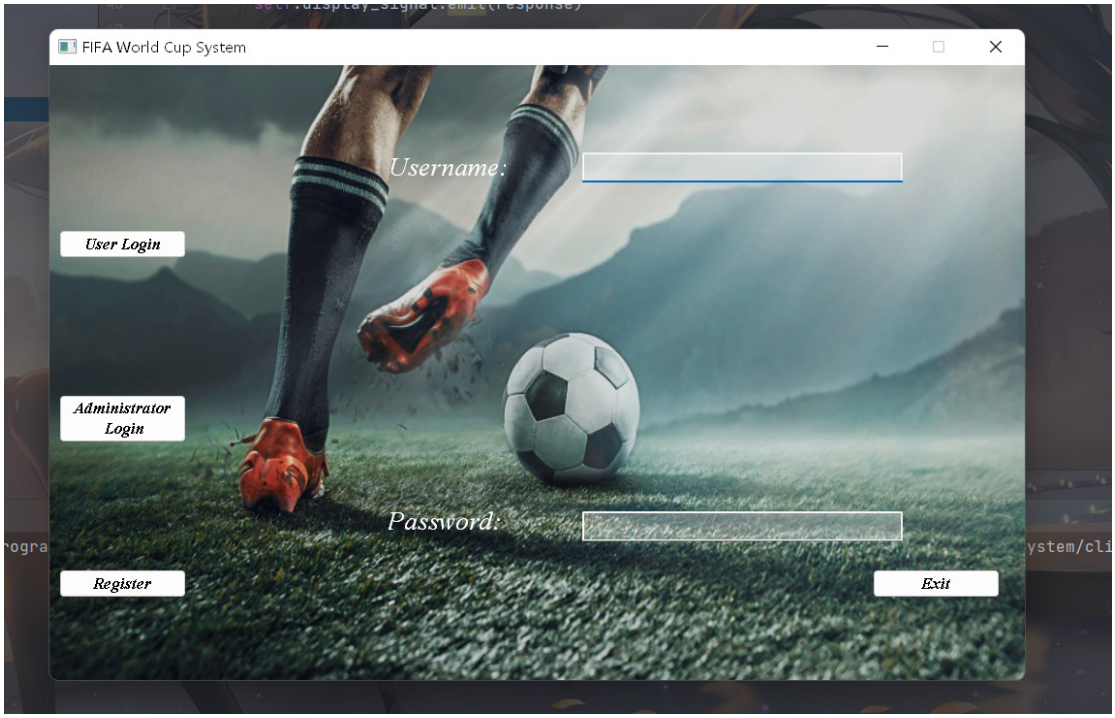


If we enter nothing

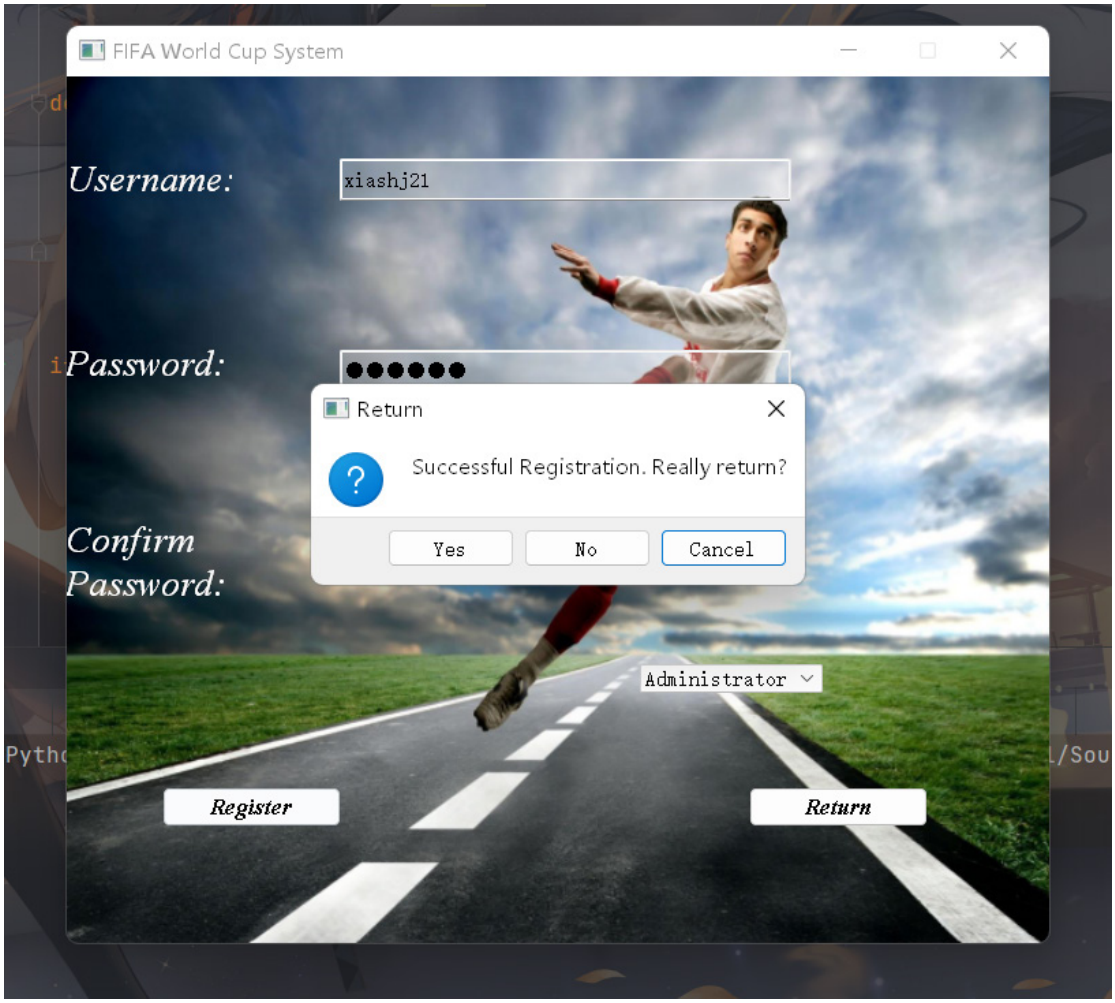


We enter 21 Nov in the Date box, 00:00 in the Time box and click on the Query button.

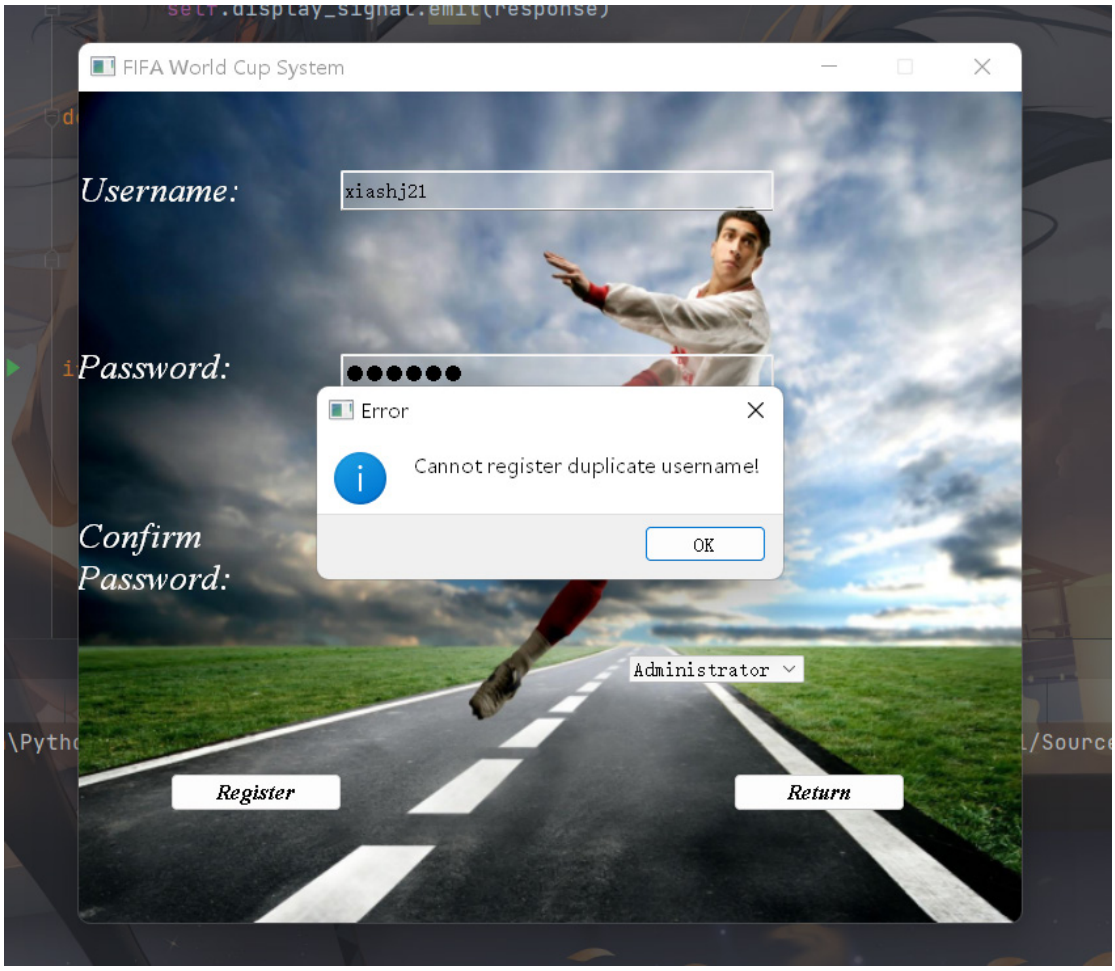
Click on the Return button.



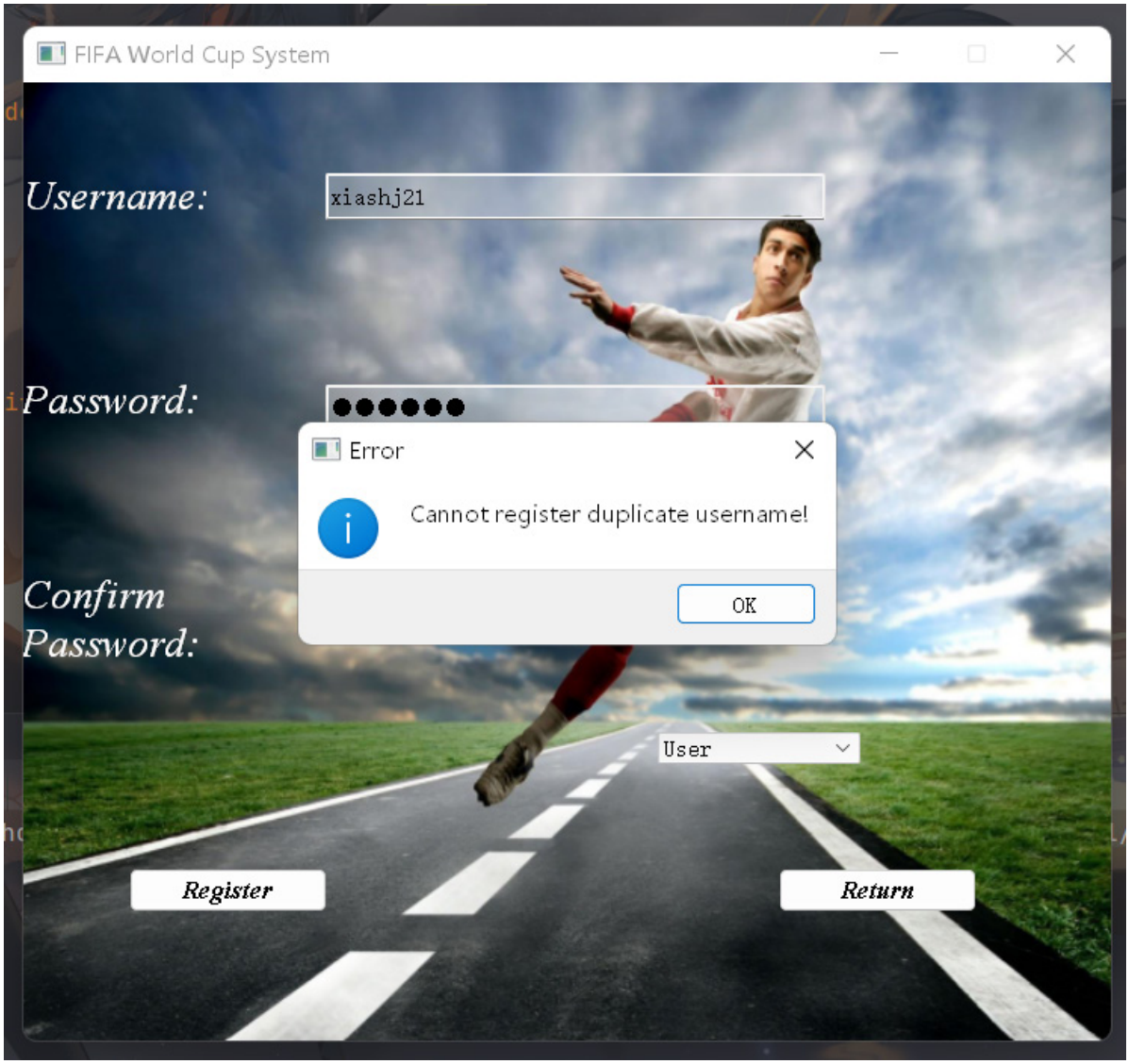
Next, we register the administrator account for user xiashj21, click the Register button, enter username xiashj21, password 123456, user type Administrator, and then click the Register button, the result is as follows.



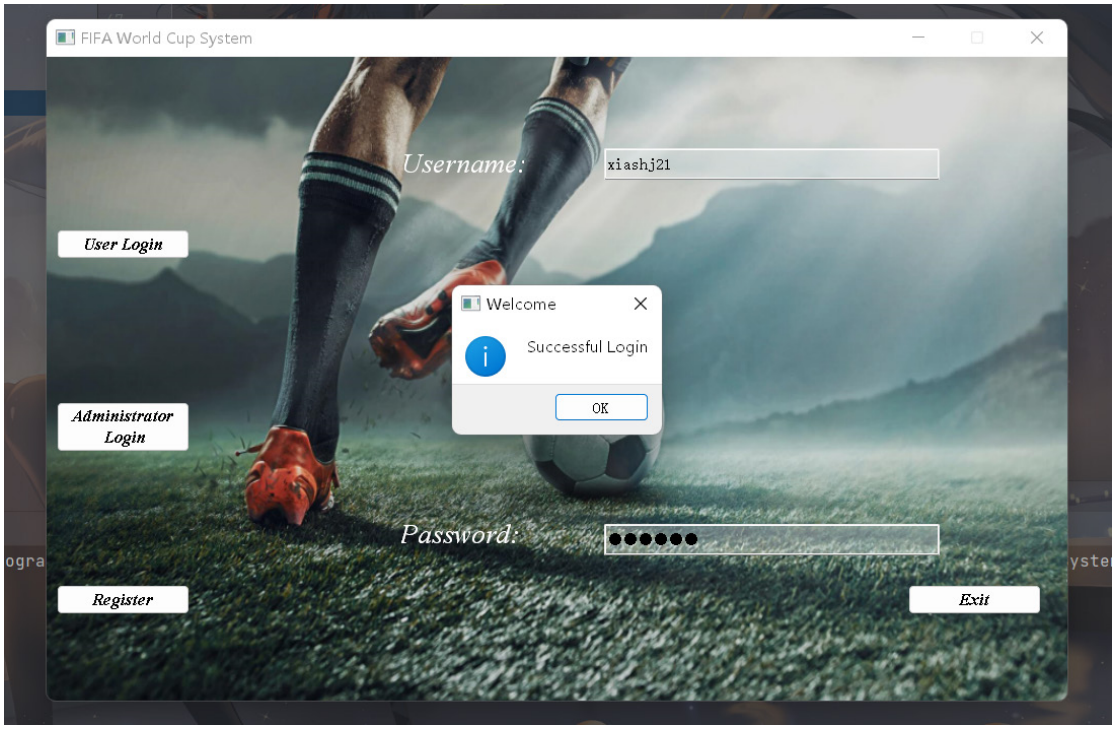
If we click Cancel and click the Register button again, we will find.



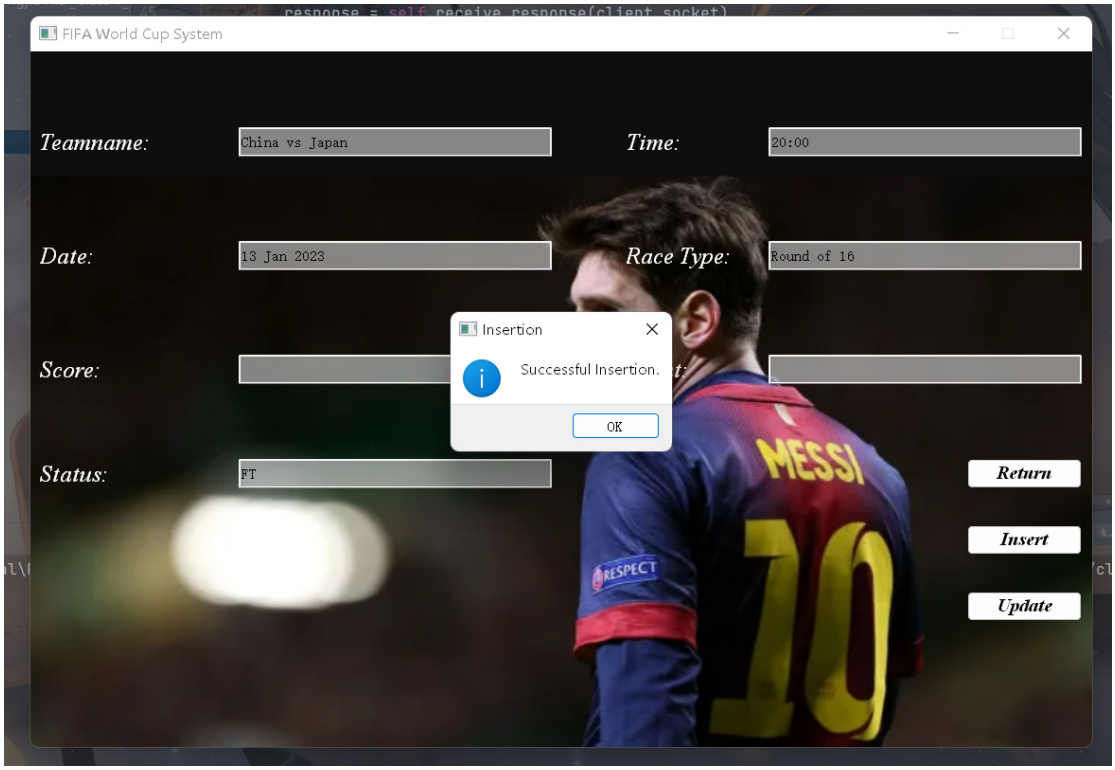
If we change the user type to normal user, we still get the same result.



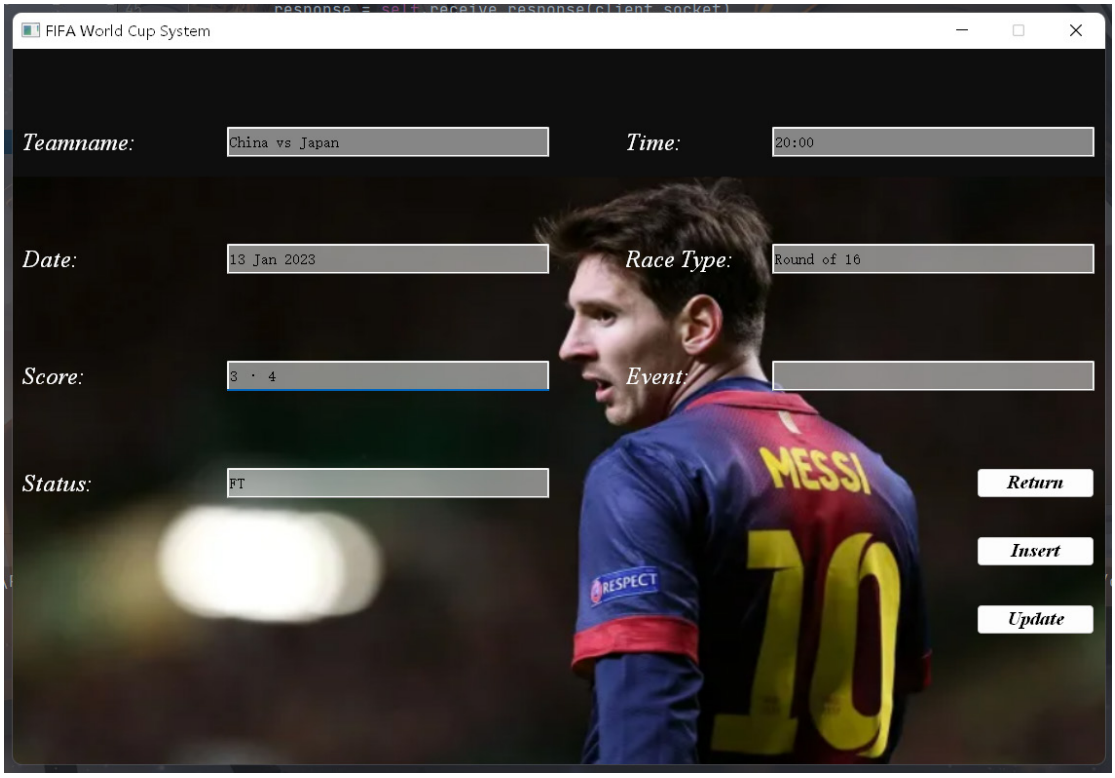
Click the Return button and make an administrator login.



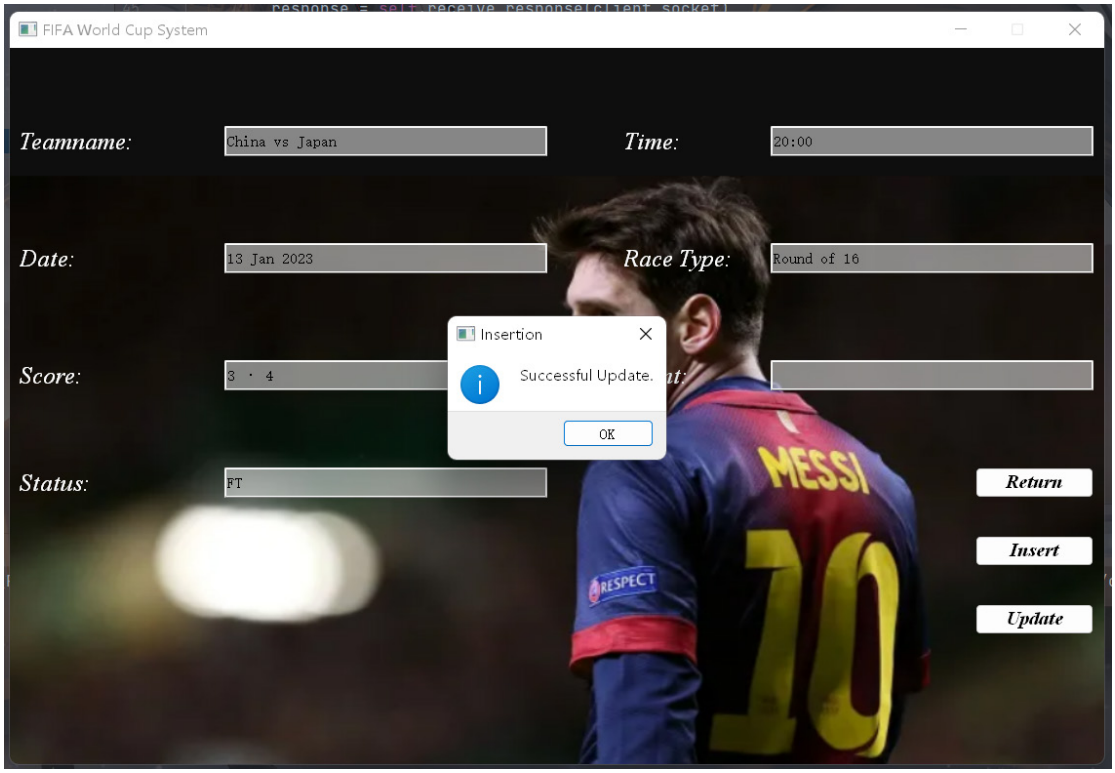
Create a new tournament.



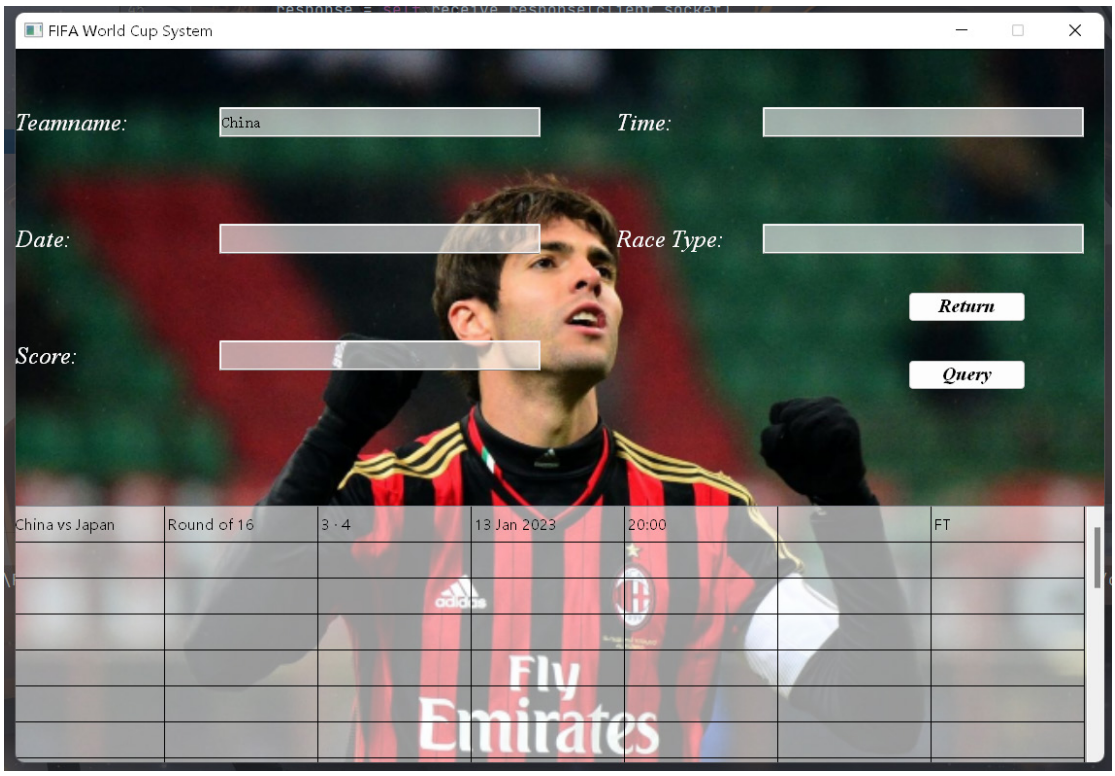
Update tournament data.



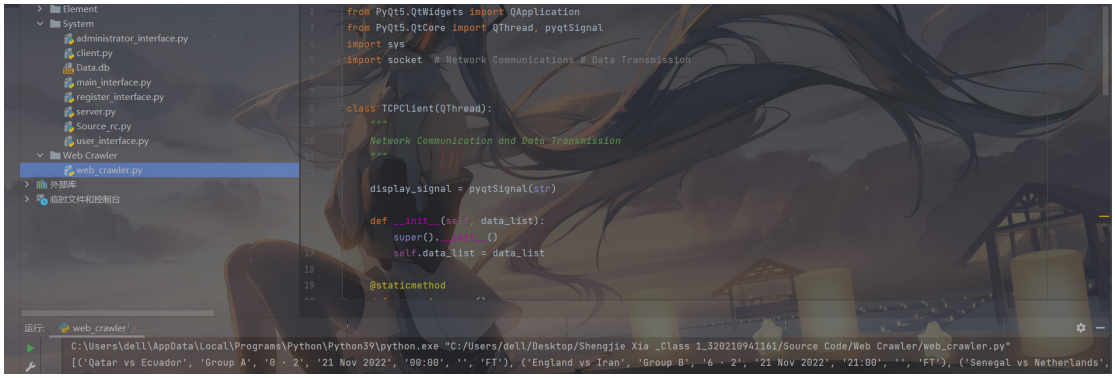
Click the Update button.



Click the Return button to log in as a normal user to query the event you just added.



The crawler runs as follows.



Instructions for Use

I am using Windows 11 as my operating system.

To ensure that the system works properly, we need to install the library PyQt5. First, make sure Python is installed on your system (the path to the Python installation needs to be added to the environment variables) I'm using Python 3.9 and Python 3.10 is also working. and then install PyQt5 using the following command from the command line.

pip install PyQt5

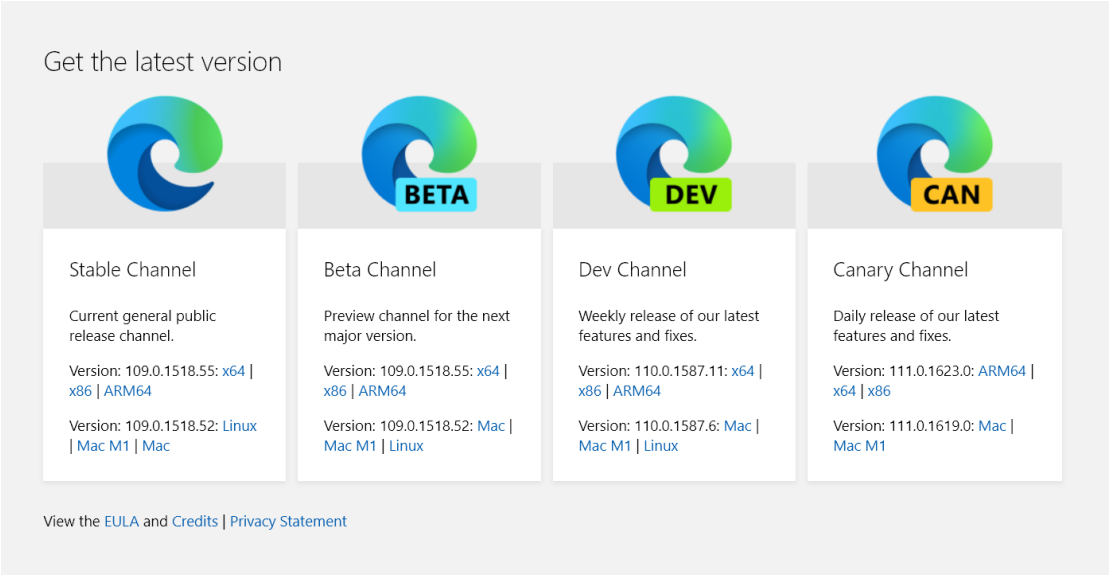
If you are installing PyQt5 in an IDE, you need to install it in the virtual environment of the IDE using the above code, or change the interpreter path to the system interpreter path.

In order to run the crawler file, we need to additionally install the following.

pip install pyautogui
pip install selenium

Then we also need to download a browser driver file, here I use the browser for Edge, so download the Edge driver. The download path for this driver is as follows.

[Microsoft Edge WebDriver - Microsoft Edge Developer]



I chose the x64 version of the driver based on my browser version and processor model (Intel processor). Download it and get the file.

After unpacking the file, you get this file.

Put them in the Scripts folder of the Python installation directory.

The name of the folder after the Users folder is the name displayed when the computer is booted.

Experience

Through this project, I initially learned about the design of graphical interface, and also learned about the knowledge related to network transmission, and reviewed multi-threaded and object-oriented programming, I think there are still many gains from it.

When doing network communication, I initially wanted to be able to transfer data types like dictionaries and lists, so that I wouldn't need to do a series of string splitting and reorganization operations. So, at first, I tried to use json library and pickle library to do so, but both ended up in failure. At first, I couldn't figure it out, but when I checked the transferred data, I found that only the first data transfer was correct after the program was running, and the data would be garbled when the page jumped or continued to transfer. I think it may be related to PyQt5 library, and also may be related to my server-side receiving function, because at that time, to prevent json or pickle receiving empty packets, I first used the socket to receive data in the front. So in the end I chose to use socket to receive data, and then do the corresponding string operation. This way the network transmission is more stable.